UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

CONSTRUCTION AND ANALYSIS OF THE TREE OF SHAPES FOR IMAGE PROCESSING

ESCOLA TÈCNICA D'ENGINYERIA DE TELECOMUNICACIÓ DE BARCELONA

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona
UNIVERSITAT POLITÈCNICA DE CATALUNYA

Tesis de Licenciatura

Grado en Sistemas Audiovisuales

Autor:
Pol Carballo i Varela

Supervisor/Tutor:
Philippe Salembier Clairon

Barcelona, Octubre 2018

Abstract

At the end of this memory we will have been developed an algorithm capable of creating a new type of representation of an image. The intention is to design an algorithm to create a hierarchical tree that orders the image by its different areas of gray level values through nodes interconnected between them by branches. A simple representation, without losses of information and without redundancies.

Similar representations that can feed this new design will be studied and then we will start the algorithm creation process.

See the results, both quality and time of calculations, will give us an approximation of its functionality and its possible applications in the image processing in the future.

Resum

Al final d' aquesta memòria s'haurà desenvolupat un algoritme capaç de crear un nou mode de representació d'una imatge. La intenció és dissenyar un algoritme capaç de crear un arbre jeràrquic que ordeni la imatge per les seves zones de valor de gris diferents a partir de nodes interconnectats mitjançant branques Una representació senzilla sense pèrdues y sense informació redundant.

S'estudiaran diferents representacions similars que poden nodrir aquest nou disseny y posteriorment s'iniciarà el procés de creació del algoritme.

Veure la qualitat dels resultats i el temps de càlcul ens donarà una aproximació de la seva funcionalitat i de les seves possibles aplicacions en el processat d'imatges.

Resumen

Al final de esta memoria se habrá desarrollado un algoritmo capaz de crear un nuevo tipo de representación de una imagen. La intención es diseñar un algoritmo capaz de crear un árbol jerárquico que ordene la imagen por sus distintas zonas de valor de gris mediante nodos interconectados entre ellos por ramas. Una representación sencilla, sin pérdida de información y sin redundancias.

Se estudiaran distintas representaciones similares que pueden alimentar este nuevo diseño y posteriormente se iniciara el proceso de creación del algoritmo.

Ver los resultados tanto de calidad como de tiempo de cálculo nos dará una aproximación de su funcionalidad y de sus posibles aplicaciones en el procesado de imágenes.

Agradecimientos

Este proyecto pone punto final a una etapa importante de mi vida y marca el inicio de la próxima.

Me gustaría tener unas palabras de gratitud para mi familia por darme la oportunidad de estar donde estoy ahora, por su apoyo y su paciencia. También agradecer a mi pareja y a mis amigos por rodearme y nuca dejarme solo. No quiero olvidarme de todos mis compañeros de Universidad que entre todos logramos llegar hasta este punto. Sin todos ellos no sería la misma persona que soy ahora.

Agradecer y gratificar al tutor y director de la tesis Philippe por su ayuda durante el proyecto y permitirme participar en este trabajo.

Finalmente quiero acordarme de todos los profesores que me han enseñado tanto durante mi estancia en la Universidad, por hacerme aprender y mejorar día a día.

Muchas gracias por todo.

Historial de revisiones

Revision	Date	Purpose
0	17/09/2018	Document creation
1	01/10/2018	Document revision
2	05/10/2018	Document revision

DOCUMENT DISTRIBUTION LIST

Name	e-mail
Estudent	
Pol Carballo i Varela	pol.carballo@alu-etsetb.upc.edu pocarva@gmail.com
Supervisor	
Philippe Salembier Clairon	philippe.salembier@upc.edu

Written by:		Reviewed and approved by:	
Date	30/09/2018	Date	05/10/2018
Name	Pol Carballo i Varela	Name	Philippe Salembier Clairon
Position	Project Author	Position	Project Supervisor

Tabla de Contenidos

Lista de Figuras

Lista de Tablas

1. Introducción

1.1. Objetivo del Proyecto

Este Proyecto fue concebido con la intención de estudiar, diseñar y generar un nuevo método de representación de imágenes mediante una estructura en forma de árbol jerárquico. La creación de este Árbol de Formas (Tree Of Shapes [ToS]) se basa en un agrupamiento de los píxeles de la imagen formando unos nuevos nodos, caracterizados por varias propiedades de los píxeles; generando una estructura de árbol interconectando estos nodos mediante ramas según la jerarquía de cada nodo en relación con todos los demás.

Para desarrollar un algoritmo generador de un ToS se aporta un estudio de otras representaciones en forma de árbol de una índole afín para intentar incluir sus fundamentos y usos al nuevo árbol a diseñar.

1.2. Requerimientos y especificaciones

- Creación del Tree Of Shapes a partir de otros árboles ya existentes (Matlab).
- Creación del Tree Of Shapes a partir de la imagen original (Matlab).
- Regeneración perfecta de la imagen a partir del árbol.
- Optimización del tiempo de cálculo.

1.3. Metodología y precedentes

El proyecto es un trabajo nuevo iniciado desde cero. Aun así se nutre de estudios realizados por otros profesores e investigadores que posteriormente se citarán en la memoria. El trabajo también emplea scripts y funciones creadas con anterioridad al proyecto por el profesor de la UPC y tutor de esta tesis, Philippe Salembier. A medida que avance esta memoria se irán disgregando y explicando estas funciones no realizadas exclusivamente para el cometido de este proyecto.

Para la confección del algoritmo debe usarse una función con el cometido de ordenar una lista de elementos. Dicha función es una adaptación del algoritmo de ordenación rápida “QuickSort o Algoritmo de Hoare” creado por el científico de computación Tony Hoare. Se considera uno de los algoritmos más eficientes y con mejores resultados para su tarea.

Todas las otras funciones no mencionadas anteriormente han sido confeccionadas exclusivamente para configurar y complementar este proyecto.

1.4. Estructura del proyecto

Este documento se estructura en estos grandes bloques:

Capítulo 2. Análisis del estado del arte en la materia y estudio más enfatizado de los árboles de máximos y mínimos, Maxtree & Mintree, para buscar posibles aplicaciones y un enfoque conjunto de ellos que pueda converger en un Tree of Shapes.

Capítulo 3. Se exponen los conceptos básicos para empezar la creación del árbol y posteriormente se analizan paso a paso los distintos algoritmos de generación del Tree of Shapes y su proceso de creación.

Capítulo 4. Análisis de los resultados obtenidos. Cualitativos y cuantitativos.

Capítulo 5. Gastos atribuidos al desarrollo del proyecto.

Capítulo 6. Conclusiones y líneas futuras del algoritmo.

1.5. Work Plan y diagrama Gantt

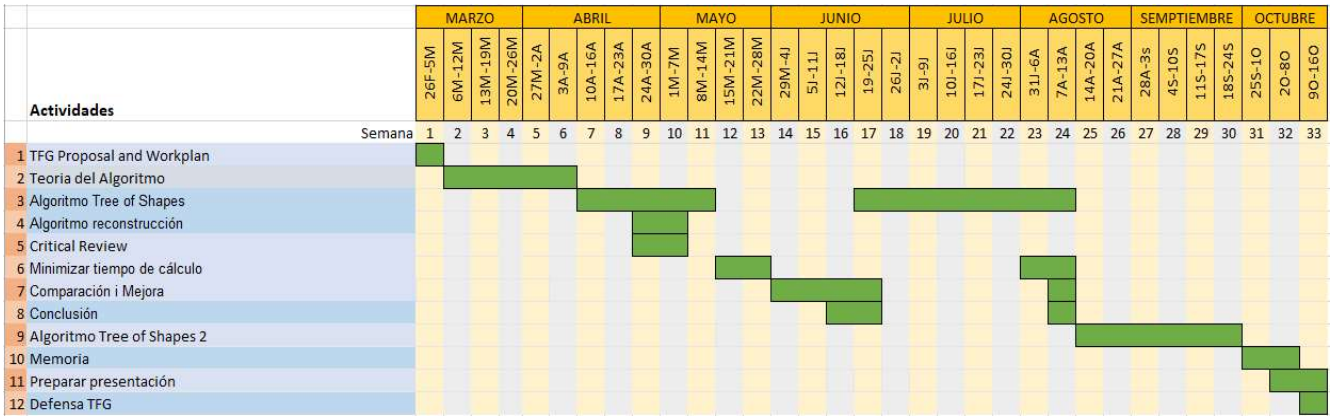

Figura 1 Diagrama Gantt del Work Plan del proyecto

1.6. Incidencias y desviaciones del planteamiento inicial

Para empezar se puede decir que la desviación principal del planteamiento fue prorrogar la entrega del proyecto de julio hasta octubre. Esta fue debida a que el algoritmo aún no estaba finalizado por complicaciones técnicas del propio algoritmo y por falta de tiempo a causa de obligaciones laborales que difieren del proyecto.

En el decurso de conseguir el algoritmo finalizado, se han descubierto errores o algoritmo no óptimos. Durante la realización se tuvo que reestructurar la idea inicial del proyecto, primero añadiendo elementos no imaginados en la fase de adquisición teórica (Maxtree y Mintree [3.2]) y posteriormente abandonando una de las ideas iniciales y generar otro algoritmo mediante otras herramientas [3.33.3].

Por ese motivo a lo largo del proyecto se han realizado tres algoritmos para generar el Tree of Shapes, el primero se modificó en un segundo algoritmo similar y posteriormente a este se generó otro totalmente distinto (visible en la Figura 1). En cada uno de estos algoritmo evidentemente también era necesario un estudio del tiempo de cálculo y como mejorarlo, y una revisión de funcionamiento correcto. Estos estudios fueron los que en la fase de conclusión ayudaron a decidir generar nuevas formas distintas de implementar el árbol.

La combinación de la falta de tiempo para una dedicación total al proyecto y los distintos inconvenientes técnicos de los algoritmos que se generaban fueron los culpables de un dilatación del WorkPlan inicial.

2. Estado del arte y metodologías similares

En los inicios de este siglo se empiezan a buscar nuevas formas de representar imágenes mediante estructuras jerárquicas para ayudar futuros procedimientos de procesado de imágenes u otras aplicaciones plausibles. Se diseñaron estructuras en forma de árbol capaces de organizar y seccionar distintas zonas de la imagen a partir de las sus zonas de valores de gris para organizarlas de un modo jerárquico dando lugar a zonas más importantes o relevantes que otras a la hora de un futurible procesado. Creando partes de una imagen dependientes y descendientes de otras partes de la imagen de jerarquía mayor.

Uno de los estudios de este tipo dio lugar al diseño de los árboles de máximos y los árboles de mínimos (Maxtree & Mintree [1]). A partir de allí se buscaron nuevas implementaciones más complejas y completas de árboles basados en una misma idea pero teniendo más en cuenta las formas aparecidas en cada imagen analizada. Empiezan a salir estudios y diseños de nuevos árboles, uno de ellos el Árbol de Formas (Tree of Shapes) y el algoritmo Fast Level Set Transform [FLST] [2], creado por el profesor Pascal Monasse.

2.1. El árbol de máximos, Maxtree

2.1.1. Significado

El árbol de máximos es una representación jerárquica, a partir de nodos y ramas, que pretende organizar las zonas planas de cada imagen a través de la relación de cada una de estas zonas con sus otras zonas conexas. El Maxtree es una partición de las zonas planas de una imagen organizadas jerárquicamente con valor de gris ascendente (los picos de valor máximo son las ramas superiores del árbol).

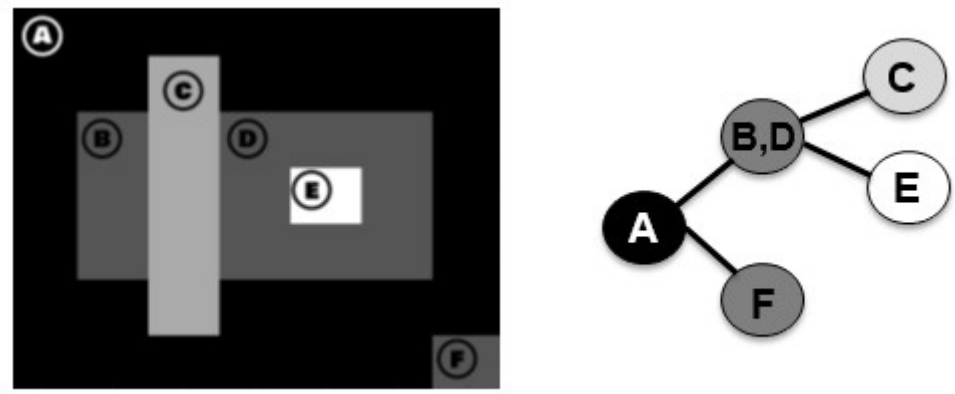

Figura 2 Representación del Maxtree

2.1.2. Proceso de creación

A continuación se sintetiza el proceso de creación del árbol de máximos a partir del artículo "*Antiextensive Connected Operators for Image and Sequence Processing*" del profesor y tutor de este proyecto Philippe Salembier en colaboración de los profesores Albert Oliveras y Luis Garrido [1].

Como visión global del algoritmo se podría decir que se crea un árbol disgregando las zonas planas de una imagen mediante un algoritmo recursivo basado en un umbral de valores de gris de los píxeles. Cada nodo del árbol corresponde a un umbral, todas las componentes conexas de un valor de gris mayor o iguales a ese umbral pertenecen al nodo. Ese es su significado físico pero en la realización del algoritmo cada nodo contiene solo pixeles del mismo valor del umbral, para evitar repetir información.

El diseño se basa de la binarización de la imagen, marcando cada píxel como inferior al umbral o igual/superior a él. Inicialmente tomamos la imagen entera I^i y aplicamos un umbral (h) al menor nivel $h = 0$. El proceso consiste en añadir todos los píxeles de la imagen de valor igual al umbral al nodo inicial del árbol, en primera instancia todos los valores de gris iguales a cero, $N^1_{h=0} = \{I^i(i,j) = (h = 0)\}$. Se crea el nodo raíz N^1_0 que consiste en todos los píxeles de la imagen I^i igual al threshold de valor 0. A su vez virtualmente se generaría un nodo (no existente realmente) con todos aquellos píxeles no contenidos en el nodo del árbol, esta separación entre píxeles superiores e iguales al umbral y píxeles inferiores a él es lo que llamamos binarización (Este nodo inicial de h=0 no tiene efecto la binarización porque todos los valores son superiores al umbral). Este nodo virtual lo almacenamos como I^1_1, que consiste en la imagen sin los valores del nodo raíz; a partir de él aplicaremos el siguiente nivel consistente a la binarización por $h = 1$.

Veamos la realización del algoritmo en el ejemplo de la Figura 3:

El primer paso es asignar el valor inicial al threshold $(h = 0)$ y ejecutar la primera binarización. Todos los valores de la imagen iguales al umbral se asignan al nodo raíz (también representa todos los valores superiores al umbral pero por motivos de ahorro de memoria no se repite la información), que en este caso corresponde a la zona plana A, $N^1_0 = \{A\}$. Con el resto de zonas o componentes de valor de gris superior al umbral aparece un caso particular, aparecen dos componentes separadas que temporalmente se asignan a $I^1_1 = \{F\}, I^2_1 = \{B, C, D, E\}$. En este momento tendríamos el primer paso realizado y por consecuencia el árbol primitivo, es momento de aumentar el umbral para la segunda iteración y aplicarlo en todos los nodos I^k_1. Haciendo el mismo proceso ahora obtenemos los nodos $N^1_1 = \{F\}, N^2_1 = \{B, D\}$, que son nodos hijos del nodo N^1_0. Del mismo modo anterior se genera el nodo temporal $I^1_2 = \{C\}, I^2_2 = \{E\}$, debido a que nuevamente tenemos dos componentes no conectadas entre sí. Las siguientes iteraciones son más sencillas, para $h = 2$ se genera el nodo $N^1_2 = \{C\}$ ya que es la única zona de este valor de gris y para el umbral a $h = 3$ se crea $N^1_3 = \{E\}$, ambos hijos de N^2_1. El siguiente valor de gris ya no está presente en la imagen, por lo consecuente se termina el algoritmo.

Posteriormente, a los nodos se les puede añadir información interesante como el número de píxeles que lo conforman. Esta información complementaria es inherente al nodo (número de nodo, padre del nodo, sus hijos y sus píxeles).

Existe la posibilidad de introducir un factor de fluctuación ($\Delta = \pm n$) para generar umbrales en intervalo para añadir más flexibilidad a la creación de zonas planas, $h = k + \Delta$. Pero en este proyecto solo nos centramos en zonas planas estrictas, de umbral constante.

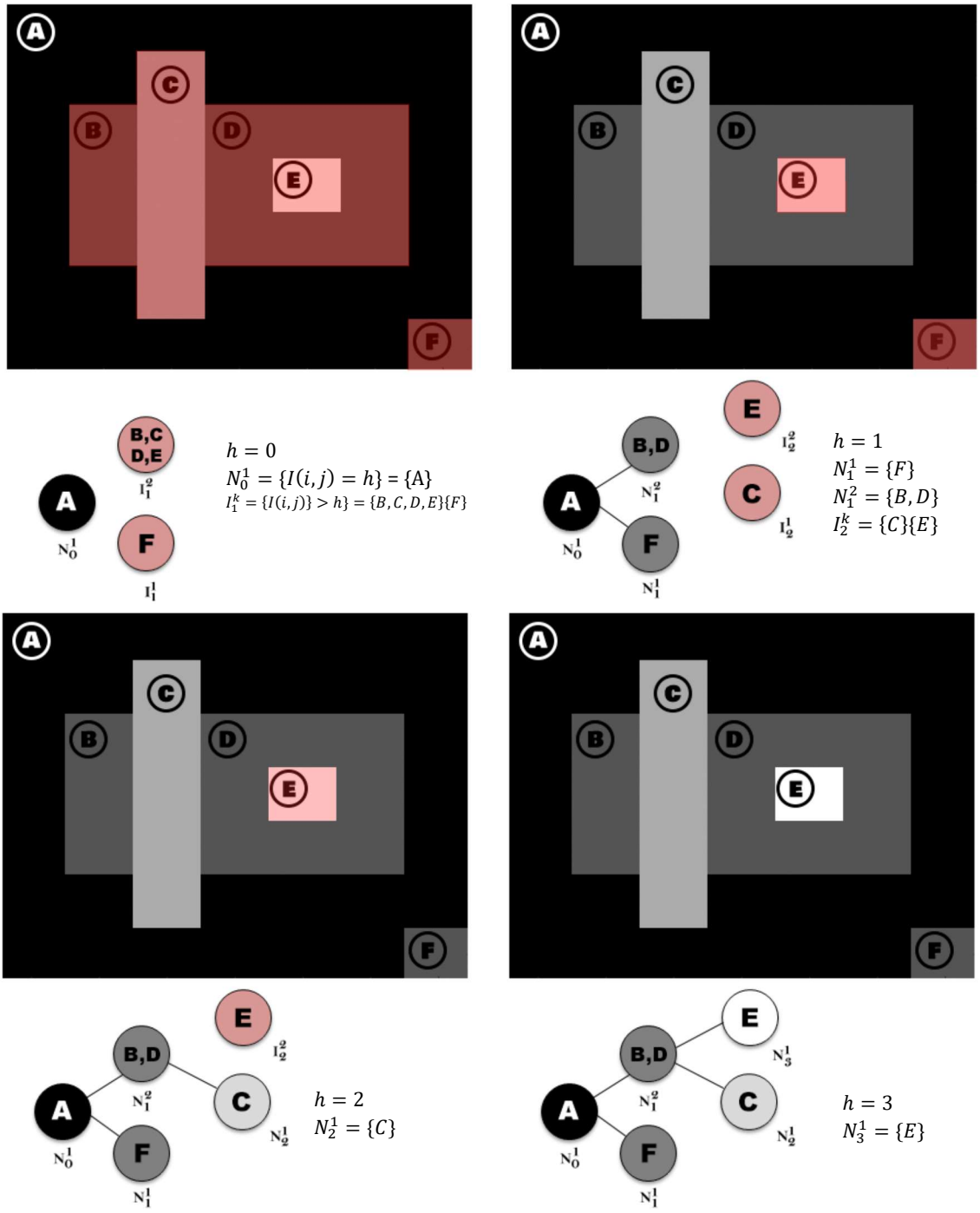

Figura 3 Proceso de creación del Maxtree

2.1.3. Carencias del Maxtree

Por definición, tal y como indica el propio nombre, el árbol de máximos o Maxtree solo enfatiza los picos de la imagen con los valores de gris más altos y los coloca en las ramas y nodos superiores del árbol. Por otra parte las zonas de valor bajo en la escala de nivel de gris quedan totalmente enmascaradas dentro del nodo raíz, pudiendo quedar escondidos picos de valores bajos; a estos picos de inferior valor también se les puede llamar agujeros. El Maxtree no aporta ninguna información sobre estos agujeros en la imagen (el árbol Mintree se encarga de estudiarlos [2.2]) y es por eso que se buscan nuevas soluciones y nuevas interpretaciones de la imagen en forma de árbol que sean más completas.

Otro inconveniente bastante notorio del Maxtree es la existencia de varias componentes, de varias zonas planas de la imagen distintas, en un mismo nodo del árbol. Este fenómeno no sucede en las ramas superiores del árbol, pero en cuanto vamos recorriendo el árbol en dirección contraria observamos que a medida que nos acercamos a los nodos de nivel de gris más bajos cada vez empiezan a contener en ellos más de una componente. En el ejemplo de la Figura 3 se puede apreciar esta problemática en el nodo $N_1^2 = \{B, D\}$, donde dos componentes distintas como son B y D forman parte del mismo nodo. Para el árbol Maxtree no aporta problemática alguna, pero si queremos crear un árbol con una separación nodal para cada zona plana de la imagen este árbol no ayuda a ese cometido en las zonas de valor de gris más bajo.

2.2. El árbol de mínimos, Mintree

Una vez analizado el algoritmo Maxtree no es difícil de predecir que estos dos algoritmos tendrán un parentesco muy próximo. El árbol de mínimos se basa en la misma idea del Maxtree pero aplicando el algoritmo al revés, centrándose en los picos de valor bajo. Por lo tanto, el Mintree es un árbol jerárquico donde como nodo raíz están los valores más altos y como ramas principales los valores mínimos de la imagen.

Para generar el Mintree se emplea el mismo algoritmo variando los umbrales. Para crear este árbol se inicializa el umbral al valor más alto que aparezca en la imagen, y cada iteración va disminuyendo este umbral en una unidad. Los píxeles de la imagen que coincidan con el valor del umbral o sean inferiores a él serán añadidos al nodo (los inferiores no se añaden pero teóricamente forman parte), también se genera el nodo temporal al cual se le aplicarán los futuros umbrales [Figura 4].

Los problemas que proporciona el Mintree son exactamente los mismos que con su homólogo de máximos. Tenemos información muy vaga sobre las zonas de valores de gris altos y puede que en los nodos que les corresponden aparezca en su interior más de una componente.

De estos defectos comunes nace una idea, aprovechar el conocimiento del Maxtree de los valores de nivel de gris elevados para complementar las carencias que tiene el Mintree sobre este aspecto. Conseguir información de los picos de valores extremos, ya sean de nivel de gris elevado o reducido, creando una mezcla entre los dos árboles, y aprovechando las zonas de cada árbol con información fiable. Esta idea se desarrollará a continuación.

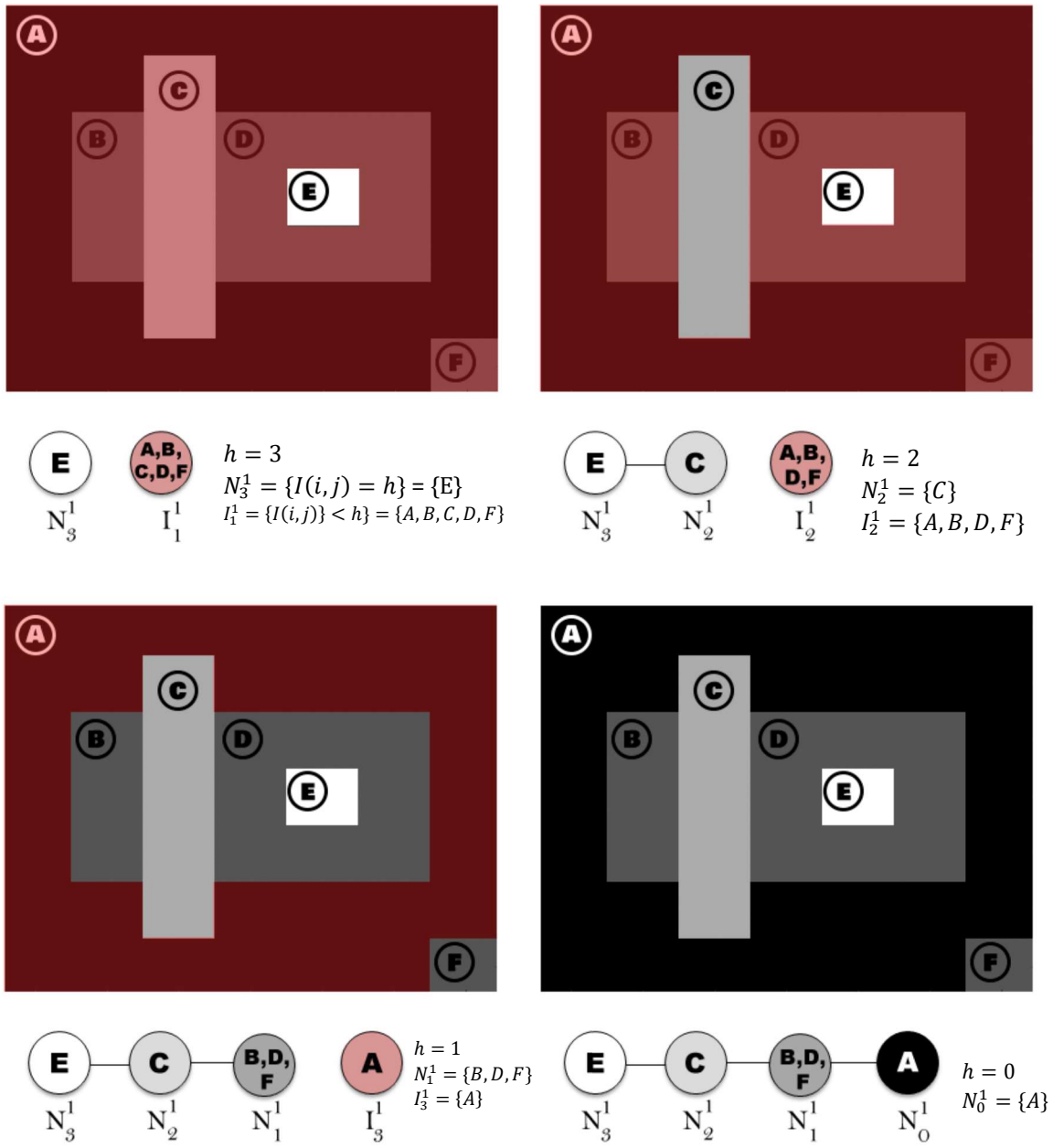

Figura 4 Proceso de creación del Mintree

2.3. Tree Of Shapes

El profesor Pascal Monasse, juntamente con el profesor Frédéric Guichard, empieza a estudiar la posibilidad de generar un árbol total que agrupe los dos tipos de árboles estudiados anteriormente. Su idea es interpretar el árbol como una estructura piramidal, donde cada nodo se analiza por su forma y su situación en la imagen, y no solo por su valor numérico.

Lo que importa en este diseño es el tamaño y la estructura de cada forma, su situación en la imagen y como está relacionada con las otras componentes que configuran la imagen. Y si estas son o no más brillantes. En este nuevo árbol toma importancia la idea de analizar para cada forma de qué otras componentes conexas está rodeada. La idea de Monasse es centrarse en las componentes que en su interior contienen y rodean otras componentes llamadas agujeros (Zonas de la imagen totalmente rodeadas por una misma componente). A las zonas de la imagen que corresponden a una componente que contiene en su interior un agujero se las puede nombrar como componentes encapsuladoras o componentes contenedoras. Cada uno de los nodos del árbol es un componente de valor de gris constante, pero este componente puede contener dentro de él otro componente encapsulado. Para generar este nuevo árbol se jerarquizan los componentes encapsuladores como nodos padre de los nodos que corresponden a la componente conexa encapsulada, agujero. Entonces se puede crear una estructura de árbol donde los descendientes de un nodo son las formas incluidas en él, y a su vez el padre es la forma más pequeña que lo contiene a él. Creando así una estructura en árbol basado en formas de la imagen con información necesaria para reconstruir la imagen a partir de él y sin información redundante.

El diseño del algoritmo empieza por la descomposición de una imagen en formas, dadas por un umbral del mismo modo que para la creación del Maxtree. Para representar cada forma de la imagen se usa el borde exterior de ella mediante una curva de Jordan que recorra dicho borde. Por lo tanto cada componente y cada nodo de la imagen está representado por una curva de Jordan, creando así una imagen únicamente de sus formas mediante la unión de todas las curvas. Cada curva pertenece a un valor de gris determinado del componente, por ese motivo podríamos afirmar que realmente es una curva de nivel de la imagen; como en los mapas topográficos. A partir de esta reflexión podríamos aseverar que el conjunto de las curvas de Jordan de la imagen formarían un mapa topográfico de la misma a partir de su nivel de gris.

- **Teorema de la curva de Jordan**: Toda curva cerrada simple divide el plano en dos componentes conexas separadas por la propia curva de frontera, la componente acotada interior a la curva y la componente no acotada exterior.

Cuando todos los componentes están caracterizados por su propia curva se busca la posición relativa de las curvas, en otras palabras, se buscan cuales curvas de Jordan son interiores a otras y por lo tanto representan un agujero. Si una curva de Jordan J_i contiene en su interior más curvas pueden suceder dos casos distintos. El primero de ellos sería que las curvas J_j y J_k están contenidas en J_i pero entre ellas no hay ninguna relación, ninguna de las dos curvas interiores contiene la otra. En este caso ambas curvas serán descendientes de la curva J_i. En un segundo caso las mismas curvas J_j y J_k sí tienen relación, por ejemplo J_j encapsula J_k. En esta situación J_i solo estaría encapsulando a la curva mayor J_j, que a su vez esta estaría encapsulando J_k. Siempre se considera el

progenitor como la forma más pequeña de contener otra forma. Finalmente el orden nodal quedaría como J_i progenitor de J_j y J_k descendiente de J_j.

- Opción 1. $J_{j,k} \in Int(J_i) \rightarrow \quad J_j \cap J_k = \emptyset$
- Opción 2. $J_{j,k} \in Int(J_i) \rightarrow \quad J_j \cap J_k = \exists \ \therefore \ J_k \in Int(J_j)$

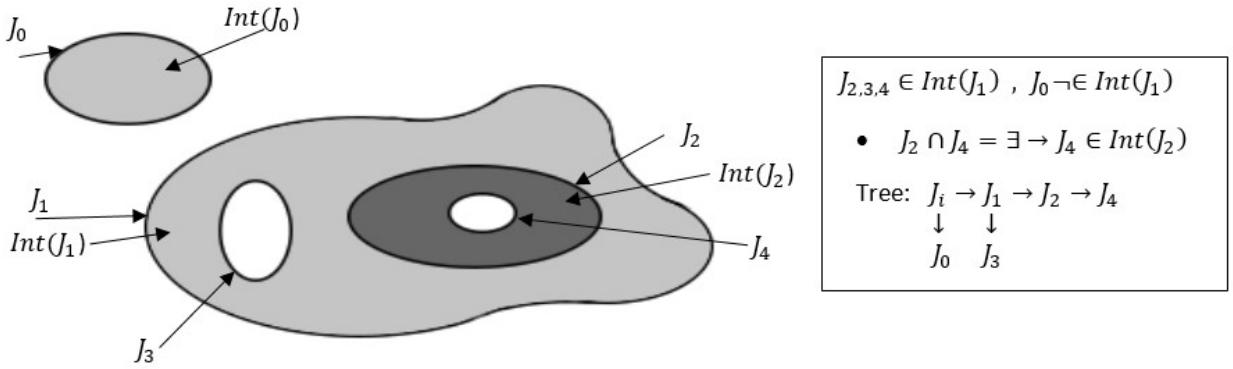

Figura 5 Análisis de jerarquía mediante curvas de Jordan

En el ejemplo [Figura 5] podemos observar como la zona blanca de la imagen contiene en su interior varias componentes ($J_{0,1,2,3,4}$). A su vez la curva J_1 contiene en su interior 3 componentes más, $J_{3,4,2}$. Por lo tanto estas últimas componentes ya no se considerarían descendientes de la zona blanca porqué esta no es la zona más pequeña que las contiene. Las curvas $J_{0,1}$ por otra parte si serian descendientes suyas. Siguiendo con el proceso, J_4 está contenida en J_2 y por lo tanto pasa a ser su descendiente y deja a las curvas $J_{3,2}$ como descendientes de su componente encapsuladora más pequeña, J_1.

La idea final de Monasse es crear el árbol de formas a partir de ir alternando el uso de Maxtree y Mintree del siguiente modo:

1. Construir el árbol de máximos Maxtree y el árbol de mínimos Mintree.
2. Buscar los agujeros en cada nodo de los árboles, mediante el sistema basado en las curvas de Jordan, y encontrar estos hoyos en los nodos correspondientes del otro árbol, y crear una relación entre ambos nodos de distinto árbol. Si no hay agujeros no hay nada que hacer.
3. Fusionar ambos árboles colocando los componentes correspondientes a los agujeros como descendientes de los que lo contienen.

A este algoritmo se lo nombra como Fast Level Lines Transform [FLLT] por el propio autor.

Posteriormente a la realización de este diseño el mismo Pascal Monasse publica su tesis doctoral basándola en el mismo objetivo. Allí analiza más a fondo los árboles de máximos y mínimos y vuelve a generar un Tree of Shapes, esta vez lo hace sin la ayuda de las curvas de Jordan y se basa en matemática de conjuntos numéricos para los píxeles de los componentes. A este nuevo algoritmo lo llama Fast Level Set Transform [FLST], pero básicamente sigue la misma estructura que en FLLT con una manera distinta de calcular y discernir si existen o no agujeros.

Este sistema ideado por Monasse, de buscar agujeros en los nodos de los árboles de extremos e ir reestructurando un nuevo árbol a partir de las coincidencias entre los nodos y los hoyos funciona, pero no muestra todas las distintas componentes conexas de una imagen por separado. Porque si los nodos analizados no tienen agujeros no son modificados. Si recordamos uno de los problemas de los árboles de extremos era la posibilidad de que estos tuvieran nodos donde convivieran más de una componente conexa, este hecho puede generar dificultades a la hora de generar el ToS para componentes conexas individuales. En este punto nos alejamos un poco de la idea de Tree of Shapes de Monasse, pero con la intención de tener, esta vez sí, un nodo para cada componente conexa de la imagen.

La intención inicial de este proyecto es trabajar en esta idea de árbol, con una metodología parecida, para poder generar un Árbol de Formas o Tree of Shapes. A continuación se detallará el proceso de creación del algoritmo del ToS, en el que se basa este proyecto, desde la idea inicial hasta el prototipo final.

3. Metodología y desarrollo del proyecto

3.1. Conceptos básicos

3.1.1. Componente

Una componente conexa es una zona donde varios píxeles vecinos de la imagen tienen un valor de gris específico e igual para todos ellos, formando una zona plana de una tonalidad de gris única. En una imagen puede haber más de una componente del mismo valor de gris siempre y cuando las dos componentes no sean vecinas, en otras palabras, que las componentes no se toquen entre ellas porque si no realmente formarían una sola componente. Cada imagen se puede fragmentar en componentes [Figura 6], equivalentes a las curvas de nivel en un mapa, donde cada píxel de la imagen está contenido en una y solo una componente conexa. A partir de estos componentes se generan los nodos del árbol, cada nodo corresponde. Como ya hemos visto para los árboles Maxtree y Mintree no siempre se cumple este requisito. Para regenerar la imagen es tan simple como volver a unir todas las componentes en su posición inicial, $I = \bigcup_i Comp_i$.

Cada componente está parametrizada por los siguientes ítems:

- **Nº de Región**. Número que identifica cada uno de los componentes.
- **GrayLevel.** Valor de gris correspondiente a la zona plana representada por el componente.
- **Pixeles.** Listado de píxeles que pertenecen a dicho componente.
- **Neighbours.** Componentes conexas que están en contacto con la componente actual, son las componentes vecinas. Si dos componentes tienen una frontera común se considera que ambas son adyacentes. Listado de números de región de los vecinos.
- **Tag.** Parámetro que sitúa cada componente en la imagen. Cada componente tiene su Tag propio. El Tag es el primer píxel de la componente que encontramos si recorremos la imagen (En Matlab la imagen se recorre columna a columna empezando por la de más a la izquierda hasta la de la derecha de arriba a abajo. Otra manera de expresar el Tag sería el número de píxeles que debe recorrer el algoritmo entre el primer píxel de la imagen hasta el primer píxel de la componente (En Matlab no existe la posición 0, el primer pixel de la imagen corresponde a la posición 1 de la imagen o matriz).
- **Done.** Campo útil para el desarrollo del proyecto, simboliza binariamente si el componente ha sido analizado o aún no.

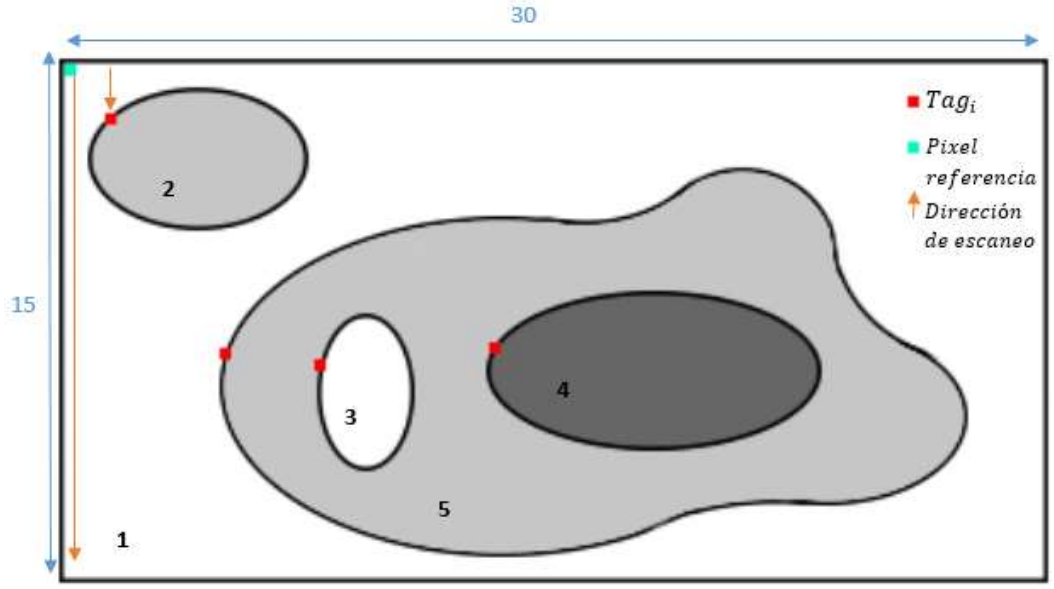

Figura 6 Componentes de una imagen con su respectivo Tag

Region	Neighbours	GrayLevel	Pixels	Tag
1	2,5	3	…	1
2	1	2	…	15+2=17
3	5	3	…	6*15+7=97
4	5	1	…	14*15+7=217
5	1,3,4	2	…	5*15+7=82

Tabla 1 Parámetros de los componentes de la Figura 5

3.1.2. Agujero

Un agujero es una zona de la imagen, puede estar formada por una o varias componentes, totalmente encapsulada por otra componente que la rodee. Es importante remarcar que para ser considerado agujero debe estar totalmente contenido dentro de la componente. Si el borde de la componente encapsuladora y el borde del agujero (Se consideran todas las componentes que conforman el agujero como una sola) se tocan en algún punto no puede considerarse agujero.

Una vez se logra identificar un agujero, las componentes que lo componen adquieren una jerarquía de descendencia de la componente encapsuladora, y análogamente la encapsuladora se convierte en nodo padre de sus agujeros.

Una componente puede encapsular varios agujeros a la vez, si estos agujero no se tocan se considera agujeros distintos dentro la misma encapsuladora.

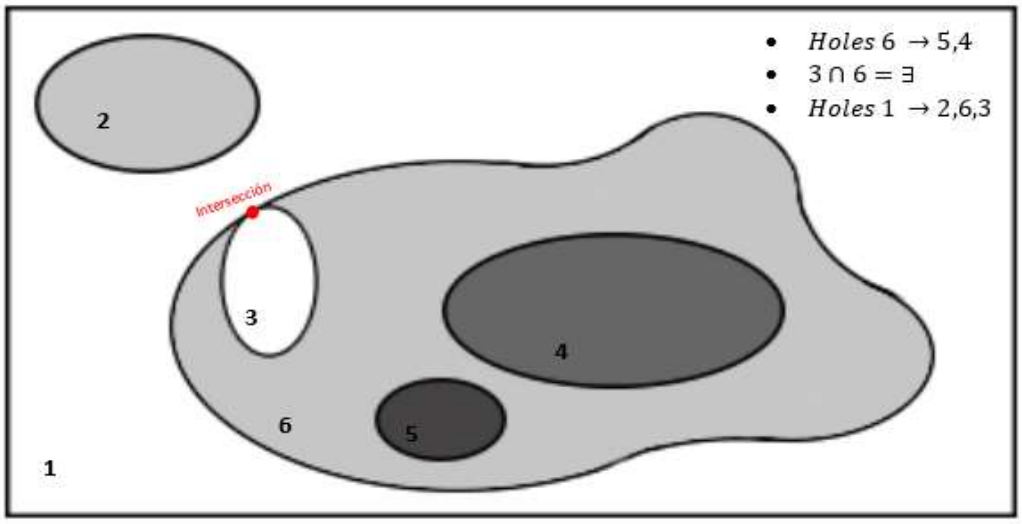

Figura 7 Búsqueda de agujeros

En el ejemplo [Figura 7] podemos observar como claramente las componentes 4 y 5 están totalmente rodeadas y contenidas en 6, por lo tanto ambas son agujeros de 6. El caso de la componente conexa 1 es parecido, claramente vemos como 2 esta encapsulada en ella. Pero tiene otro agujero, en gran medida formado por el componente 6, donde las componentes 3 y 6 tienen una intersección de sus bordes. Por lo tanto 3 no se puede considerar agujero de 6, porque no está totalmente contenido en él. Realmente el caso que nos ocupa es un ejemplo de como más de un componente pueden formar un agujero. En este caso el agujero es la unión entre 6 y 3. Finalmente obtenemos que los siguientes agujeros, $Holes\ 6 \rightarrow 5{,}4$ & $Holes\ 1 \rightarrow 2, 3 \cup 6$.

3.1.3. Conectividad

El concepto de conectividad juega un papel clave para la creación de componentes. La conectividad dicta el modo en el que se decide cómo se establecen las relaciones de adyacencia o vecindad entre pixeles. La conectividad puede fijarse en 4 direcciones (con=4) que incluyen todas las direcciones perpendiculares al pixel analizado, por lo tanto se consideran píxeles vecinos a otro los que se sitúan justo encima, abajo a izquierda y derecha. La otra modalidad de conectividad se establece en todas las 8 direcciones posibles (con=8), las cuatro anteriores añadiendo las 4 posiciones diagonales al pixel analizado [Figura 8].

Este factor es muy importante. Una conectividad de 4 puede generar bastantes más componentes que una conectividad completa de 8. No solo afecta a la confección de las componentes sino también en la relación entre ellas. Dos componentes que en conectividad 8 son vecinas podrían dejar de serlo con la conectividad 4 dependiendo de su relación relativa. Este hecho puede afectar en la confección del árbol, dejando un resultado muy distinto dependiendo del factor de conectividad.

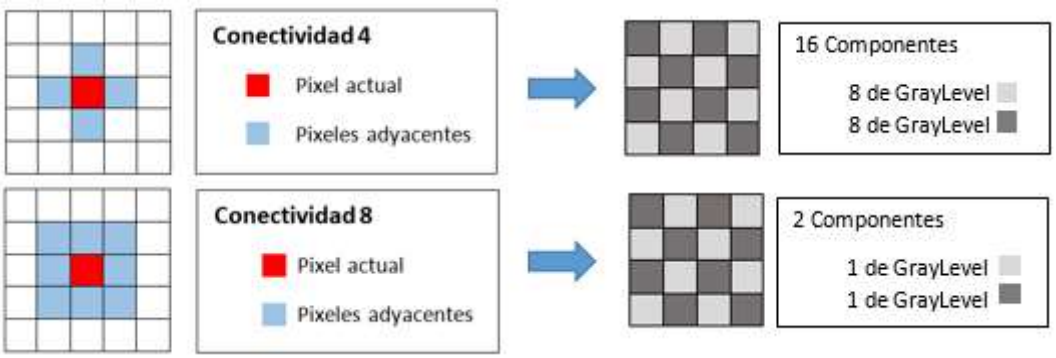

Figura 8 Significado y consecuencias de la conectividad

3.2. Proceso de creación del Algoritmo

El primer instinto es intentar generar el árbol de formas de un modo similar al expuesto anteriormente tal y como lo hizo el profesor Monasse. Por lo tanto las primeras versiones del algoritmo trabajaban es esta línea de trabajo.

La idea era simple, a partir de los árboles Maxtree y Mintree formar por si solos un Tree of Shapes sin información redundante y que fuera una representación del total de la imagen capaz de regenerar la imagen exacta [Figura 9].

Para realizar este primer diseño los pasos eran claros, se debía crear un script capaz de encontrar y reconocer agujeros a partir de un nodo de los árboles de extremos, únicamente con sus píxeles. Posteriormente, si se encontraba un "hole" en uno de los árboles había que buscar este agujero en alguna de las ramas del árbol contrario. El algoritmo funcionaba de la siguiente forma:

- Empezando por los picos del Mintree (Picos de valor de gris bajo) hasta terminar en la raíz se va recorriendo el árbol nodo a nodo.
- Se busca si hay agujeros en la componente que corresponde al nodo.
  - Sí – Si se encuentra algún agujero se analiza y se busca dicho hoyo como nodo o alguna de las ramas del otro árbol, hasta encontrar qué nodos corresponden a esa zona de la imagen. Si el nodo donde hemos encontrado el agujero es un nodo del Maxtree se busca su hole en el Mintree, homólogamente si el hoyo es encontrado en un nodo del Mintree sus componentes se buscan en el Maxtree. A este nuevo nodo encontrado le volvemos a buscar posibles agujeros y repetimos el proceso.
  - No – Se hace un recuento de los nodos analizados y añadidos al ToS para que no vuelvan a ser analizados.
- Se considera totalmente analizado el nodo y se recorre hasta el siguiente nodo del Mintree que no haya sido analizado aun.

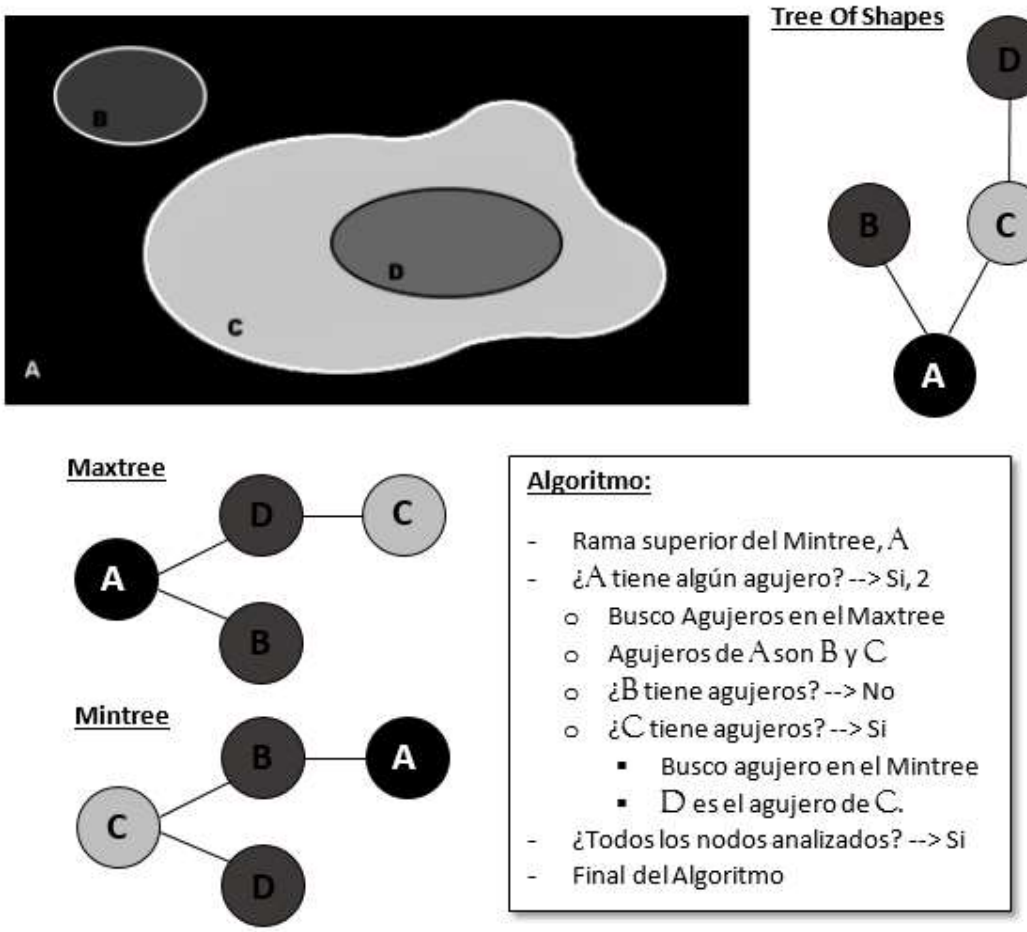

Figura 9 Algoritmo de Tree of Shapes mediante encadenado de Agujeros

Pronto se descubrió insalvable el hecho de que en los nodos de estos árboles coexisten varias componentes conexas a la vez. Este hecho generaba muchos problemas en la búsqueda de los nodos que correspondían a agujeros encontrados, ya que estos nodos podían pertenecer y no pertenecer al hole al mismo tiempo. La única solución partía de crear una segmentación para cada nodo para provocar que cada nodo correspondiese a una sola zona plana de la imagen.

De esta problemática nace un nuevo diseño basado en los árboles Maxtree y Mintree pero de un modo más indirecto que el algoritmo anterior, se precisa de un nodo para cada componente y así evitar estos errores. Para poder llevar a cabo este algoritmo se requiere una importante fase de pre-procesado que necesita un tiempo de cálculo elevado.

Una parte importante del pre-procesado se basa en la adecuación del Maxtree y Mintree para añadir algunos parámetros a los ya existentes (Píxeles, GrayLevel, Parent, Children). Usando la función *add_tree_Tags* se añaden nuevos campos de información a los árboles. Se añade un campo con los pixeles que no han sido analizados (UsePixels), un indicador de nodo analizado (Done) y el pixel referencia de cada componente conexa del nodo (Tag). Como los nodos del Maxtree/Mintree pueden contener a distintas componentes conexas, un mismo nodo puede tener varios Tags, uno por cada componente independiente. Para definir las distintas componentes conexas se usa la función bwlabel, mediante una máscara de todos los pixeles de un nodo del Maxtree, la función etiqueta cada una de las

componentes que están separadas con un número. De aquí extraemos los distintos Tags de cada componente. En este punto no se busca separar por componentes cada nodo porque se perdería la estructura del árbol, este hecho provoca que más adelante se tenga que volver a emplear un sistema similar para esa vez si, separar por componentes. Ahora solo queremos saber las distintas referencias Tag que tienen los nodos sin perder el Maxtree. El pixel referencia es el pixel más próximo al pixel inicial al recorrer la imagen. Por otra parte se debe añadir una posición a cada Pixel de la lista (+1) porqué Matlab usa como referencia inicial de una imagen el valor 1, a diferencia del valor 0 de otros lenguajes [Figura 10].

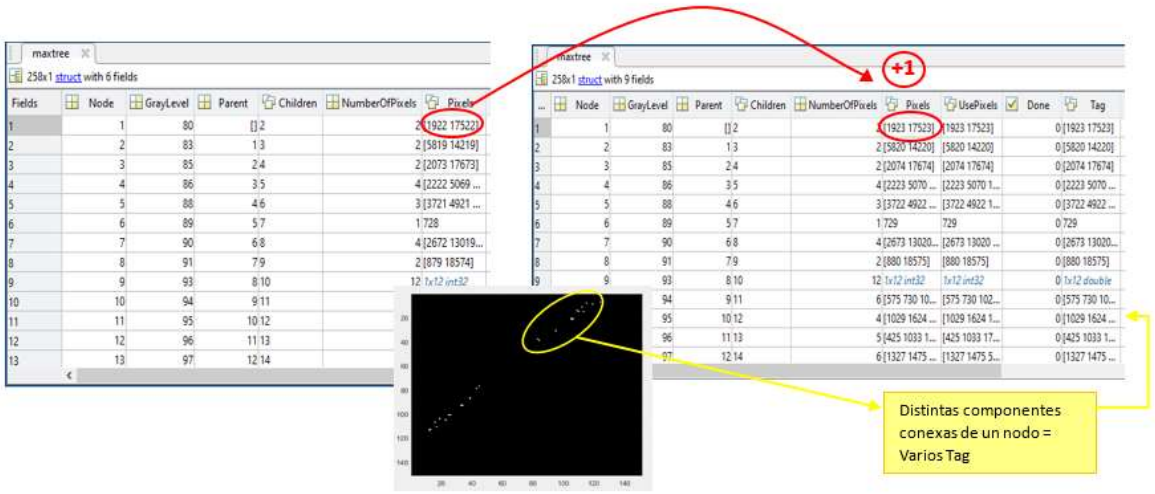

Figura 10 Parámetros añadidos al Maxtree mediante la función add_tree_tags

El pre-procesado también necesita crear una lista de todas las componentes que forman la imagen y todas sus relaciones de vecindario. Esta se llama Lista de Adyacencia (Adj) y enumera todas las componentes de la imagen e indica su valor de gris, sus píxeles y las componentes conexas adyacentes a ella (*Su funcionamiento se detallará más adelante [3.3.1]*). En este algoritmo, esta lista Adj básicamente se usa para crear algunas de las relaciones de parentesco entre nodos.

Una vez realizado el procesado previo al algoritmo, se procedió con el diseño para la creación de ToS pero tampoco ofrecía mucha simplicidad. A continuación hay una pequeña explicación esquemática de su funcionamiento.

El algoritmo pretende nutrirse de los árboles Maxtree y Mintree para crear el Tree of Shapes. Para realizar dicho cometido se recorren todos los nodos del Maxtree en dirección ascendente (del último nodo al segundo) o lo que es lo mismo, empezando por las componentes de mayor nivel de gris situadas en las ramas finales e ir en dirección hacia el nodo raíz. El último nodo del Maxtree no se analiza ya que sus componentes se analizan en el Mintree, donde serán extremos del árbol de mínimos. De este modo se analiza nodo a nodo el Maxtree. La idea es intercambiar el estudio de los árboles. Si empezamos en el Maxtree buscaremos sus agujeros en el Mintree y viceversa. El primer paso es separar las distintas componentes que conforman el nodo (puede ser que esté formado por más de

una componente). Recordemos que anteriormente usamos la función *bwlabel* para obtener los Tags de cada componente. Pues en este punto la volvemos a usar con la peculiaridad de que ahora si queremos separar cada componente en nodos distintos. Para ello creamos una máscara auxiliar (*mask_maxtree*) que se entrega a la función *bwlabel* para que etiquete cada una de las distintas componentes con un número identificador. Con ese etiquetado se crea una lista de los componentes (*components*) con la información de sus pixeles, Tag y Nivel de Gris. Una vez encontrados se aplica el siguiente algoritmo a cada uno de los componentes del mismo nodo por separado (No importa el orden porque al ser del mismo nodo tienen el mismo valor de gris):

- Se añaden los píxeles de la componente a la máscara total (*mask_total*) que indica todas las zonas de la imagen que han sido analizadas
- Se marca en el Maxtree y Mintree los píxeles que han sido analizados.
- Se añade la componente como un nuevo nodo del árbol de formas *Tree Of Shapes*
- Se buscan agujeros dentro de la componente.
- Se establece el parentesco del nuevo nodo.
- Pasar a la siguiente componente. Si no hay siguiente componente perteneciente al nodo pasar al siguiente nodo del Maxtree.

Una vez procesados todos los nodos del Maxtree excepto el primero, se procede a analizar los nodos de Mintree cuyo GrayLevel es el mínimo aparecido en la imagen. A cada nodo del Mintree de nivel mínimo se le aplica el mismo algoritmo (máscara, actualizar árboles, añadir a ToS y buscar agujeros y establecer jerarquía). En este punto ya está creado el Tree of Shapes y terminado el proceso.

Una vez finalizado el algoritmo se observa que no es eficiente en ningún aspecto. Segmentar los nodos en varias ocasiones durante el algoritmo retrasa mucho el tiempo de ejecución (En pre-procesado y durante el algoritmo). La intención de usar los árboles Maxtree y Mintree, para poder generar el Tree of Shapes, nos ha alejado mediante sus inconvenientes encontrados a medida que se avanzaba en el proyecto del funcionamiento óptimo del diseño general. Para poder catalogar correctamente cada componente conexa de los nodos multi-componente se usa la Lista de Adyacencia, donde aparecen todos los componentes de la imagen y sus vecinos o componentes adyacentes. No parece descabellado pensar que si ya tenemos una lista con todos los componentes de la imagen y todas sus posibles relaciones jerárquicas (El nodo de una componente conexa solo puede tener una relación de parentesco con algún otro nodo que corresponda a una componente que sea vecina a ella) no es necesario usar un algoritmo tan complicado para generar el árbol de formas. Si en lugar de usar la Lista de Adyacencia (*Adj*) como complemento para solucionar carencias del Maxtree y Mintree se pudiese usar por si sola para generar el Tree of Shapes podría significar un diseño más simple y rápido.

Con esta ida se realiza el diseño definitivo de algoritmo para generar de Tree of Shapes. Basándose en *Adj* y dejando de lado los árboles de máximos y mínimos se consigue un diseño más simple, menos intrincado y menos caótico a la par que un poco más veloz respecto al tiempo de cálculo.

3.3. Algoritmo de creación del Tree of Shapes

En este apartado se detalla el diseño del algoritmo implementado para generar un Tree of Shapes a partir de una imagen. Este diseño tiene como sus partes principales el pre-procesado necesario para realizar el algoritmo, el algoritmo general, la función dedicada a buscar los agujeros y como decidir la jerarquía y parentesco de los nodos del ToS. A continuación se precisan y definen estas funciones para entender qué hacen y con qué metodología lo hacen posible.

3.3.1. Pre-Procesado (preprocessed)

Este es el primer bloque del algoritmo. Es el encargado de generar las herramientas necesarias, a partir de la imagen original, para que el algoritmo se pueda llevar a cabo.

Cálculo de la matriz de adyacencia (*Adj*). En primera instancia se usa la función *cclabel* (cedida por la UPC) que asigna un valor a cada zona plana distinta de la imagen, etiqueta con un valor numérico cada componente conexa. Genera la imagen Icc.

Ejemplo de funcionamiento del *cclabel* [Figura 11]:

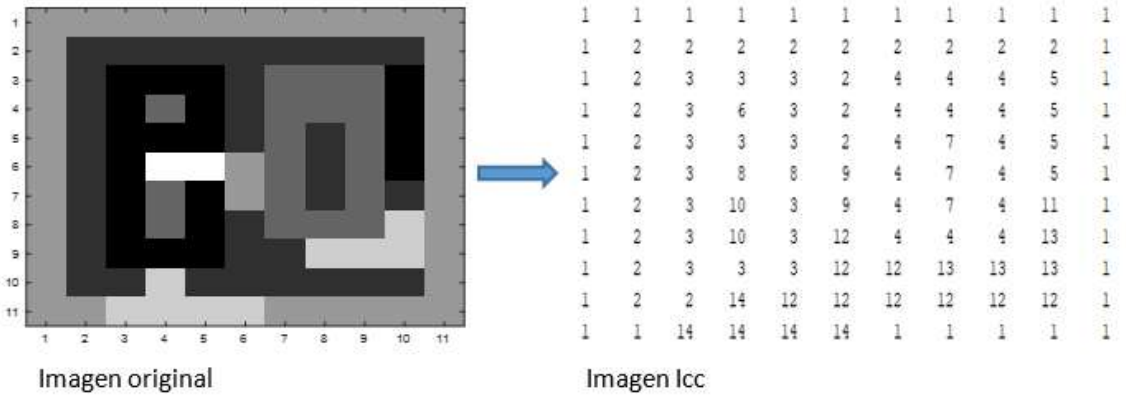

Figura 11 Funcionamiento de la función cclabel

Cada zona encontrada corresponde a una componente conexa de la imagen. Usando la función *Adjacency* es posible mostrar cómo se relacionan entre ellas las componentes, sus relaciones de vecindario. Su funcionalidad es crear una matriz de adyacencia entre todas las distintas componentes de la imagen. Cada columna representa una componente y las filas su relación con las otras componentes. Si ambas componentes (columna, fila) son vecinas la matriz indica un 1, contrariamente un 0 si no lo son. Esta matriz de adyacencia es por definición simétrica (la matriz es igual a su matriz traspuesta) y de diagonal con valor nulo, ya que ninguna componente puede ser vecina de sí misma. Si una componente es vecina de otra, esta segunda también lo es de la primera. Finalmente la matriz se traduce a una lista de regiones y todos sus vecinos para facilitar la lectura, esta lista es la Lista de Adyacencia, *Adj* en el algoritmo [Figura 12].

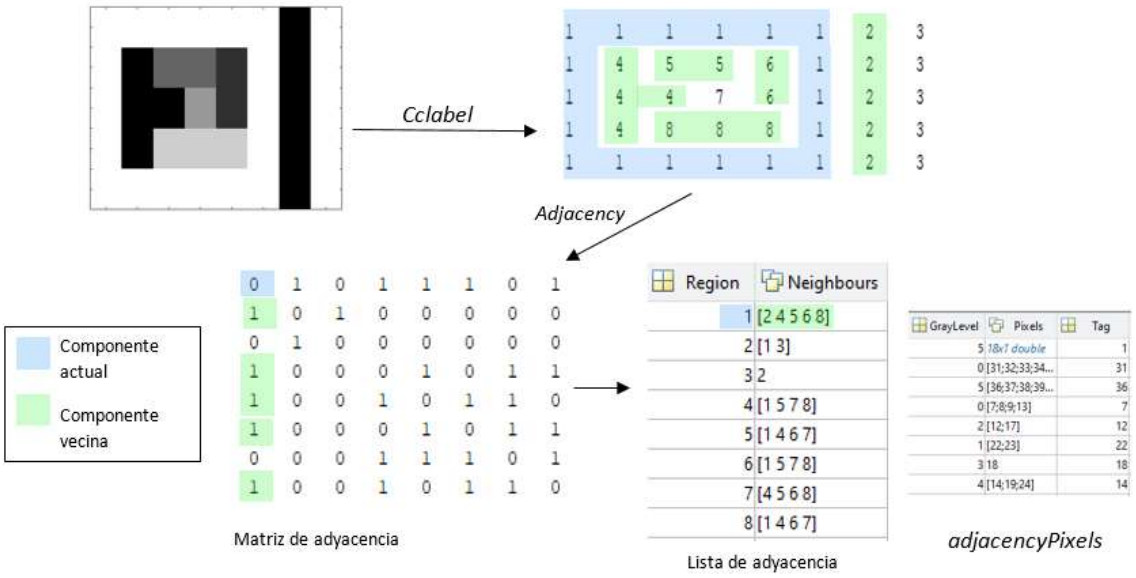

Figura 12 Proceso de creación de la Lista de Adyacencia

Para terminar se usa la función ***adjacencyPixels*** para añadir información de cada una de las componentes conexas que conforman la imagen. La información del GrayLevel de cada región es añadida a la Lista de Adyacencia, juntamente con sus píxeles y el Tag correspondiente.

Creación de dos máscaras (mask_total, mask) de valor cero del mismo tamaño que la imagen a procesar (I). Estas máscaras indicarán cuales son los píxeles de la imagen que han sido analizados. (Analizado=1, No-analizado=0).

Ordenación por nivel de gris de la Lista de Adyacencia. Para la creación del Tree of Shapes es necesario que se analicen los nodos con un orden relacionado con el nivel de gris de cada una de las zonas de la imagen. Con el fin de ordenar la lista se usa un algoritmo de ordenación llamado ***Quick Sort***. Dicho algoritmo crea un vector de nodos de la Lista de Adyacencia ordenados de mayor a menor por su nivel de gris.

3.3.1.1. Quick Sort

Vec --> Vector a ordenar.

Gray_level --> Vector que contiene el nivel de gris de los nodos del vector *vec*.

Esta función está basada en el algoritmo de ordenación rápida de listas Quick_Sort adaptado para ordenar un vector de nodos (vector original, *vec= 1, 2, 3… n*) en correspondencia al valor de gris de cada uno de los nodos (vector *gray_level*). El algoritmo ordena *gray_level* para a su vez ordenar *vec*. La función retorna los nodos de *vec* ordenados de manera que sus niveles de gris son ascendentes.

El motivo de crear esta función pudiendo usar funciones ya implementadas en Matlab (como ***sort***) es la problemática que genera el doble vector y tener que ordenar uno en función de otro. El algoritmo se aplica al vector de niveles de gris para ordenarlos ascendentemente, pero al mismo tiempo cada valor de gris corresponde a un nodo concreto de la lista *vec,* que es el vector que realmente queremos ordenar. Ambos

vectores están conectados, la posición *vec(i)* del vector corresponde a la posición *gray_level(i)* (comparten los índices *i* del vector), donde estará el valor de gris del nodo que representa *vec*. Por lo tanto, si mientras ordenamos *gray_level* duplicamos sus movimientos en *vec* ordenaremos ambos vectores. Como la función de Matlab no está diseñada para este propósito lo más eficiente era crear esta adaptación del modelo Quick_Sort.

Este es el algoritmo recursivo Quick_Sort:

- Se elige un elemento del vector que se usará como pivote referencia. En este caso se ha escogido el primer valor del vector a ordenar para mayor comodidad. La elección del pivote puede mejorar i optimizar la rapidez del procedimiento.
- Seguidamente se sitúan los demás elementos a izquierda y derecha del pivote dependiendo de si son mayores o menores que el valor del pivote. A partir de aquí el pivote ocupa la posición final que le corresponde en la lista ordenada.
- Posteriormente la lista queda dividida en dos sublistas, la de los elementos a la derecha del pivote y la de los elementos a su izquierda.
- Finalmente se repite el procedimiento para cada una de las sublistas hasta que estas estén formadas únicamente por un solo elemento.

Cuando se ordena el vector *gray_level* como la posición de cada valor corresponde al nodo indicado en la misma posición del vector *vec*, haciendo los mismos intercambios en *vec* que en *gray_level* se logra ordenar ambos. En la Figura 13 vemos esta relación entre ambos vectores durante la ordenación. Donde los desplazamientos para ordenar ascendentemente el vector de valores de gris (izquierda de la imagen) se traducen en un desplazamiento identico en el vector de nodos (derechade la imagen).

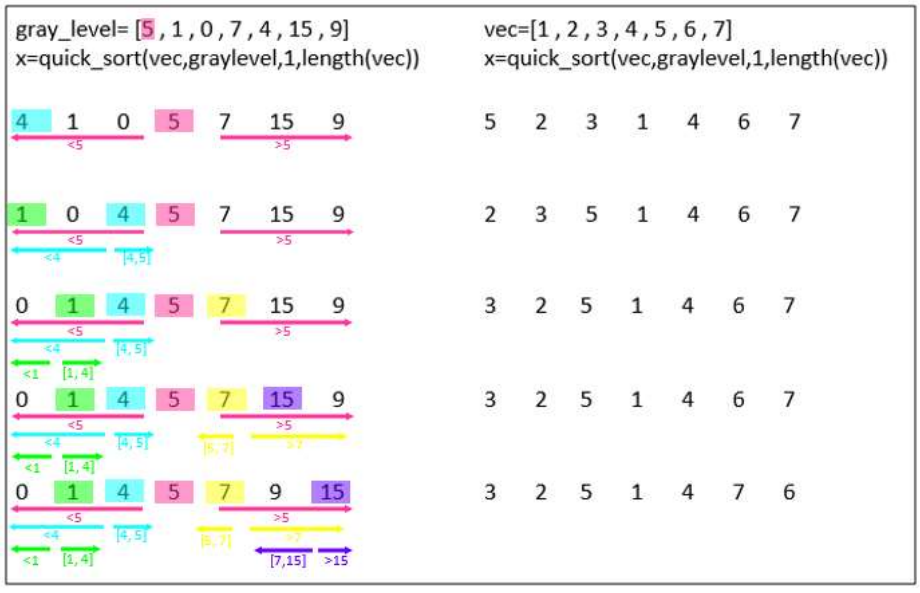

Figura 13 Algoritmo de ordenación Quick Sort

3.3.2. Buscar Agujeros (search_holes)

La finalidad de esta función es buscar si un nodo del Tree Of Shapes contiene algún agujero (otras componentes de la imagen totalmente rodeadas por la componente actual). Un mismo nodo puede tener varios agujeros distintos contenidos en él.

Búsqueda de la existencia de agujeros. Usando la función de Matlab *imfill* se rellenan los agujeros que pueda tener la máscara del componente que estamos estudiando (*mask*). Si a la nueva imagen con agujeros rellenados se le substrae la máscara del componente obtendremos una máscara únicamente de los agujeros [Figura 14]. Seguidamente se usa la función de Matlab *bwlabel* para etiquetar los posibles agujeros y se crea una lista con los pixeles y Tag (pixel más próximo al pixel inicial de la imagen) correspondientes de cada uno de los agujeros. Si no se encuentra ningún agujero la función llega a su fin, si no, sigue su curso.

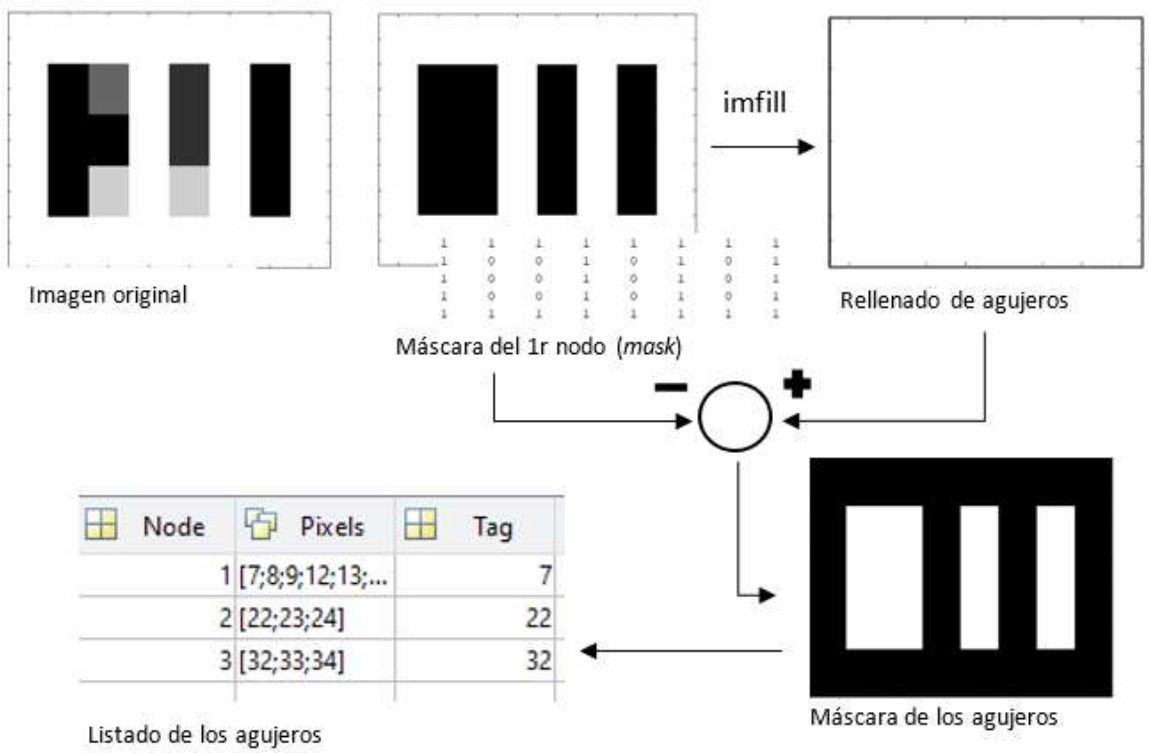

Figura 14 Funcionamiento de la búsqueda de agujeros

Si se encuentran agujeros el siguiente paso es crear una máscara individual del agujero (*mask_hole*). El algoritmo funciona agujero a agujero, lo que es lo mismo, tomando cada uno individualmente.

Una vez hemos encontrado agujeros hay que asociarlo a una o varias componentes conexas y posteriormente crear los parentescos para la creación de ToS.

Buscar parentesco de los agujeros. La búsqueda para encontrar las componentes asociadas a los agujeros empieza en el Tree Of Shapes para ver si el componente del hole ya ha sido analizado anteriormente [Figura 15]. La búsqueda funciona mediante los pixeles

del agujero, se busca algún nodo del árbol ToS que tenga los mismos pixeles. Lo que ahora se intenta es encontrar el agujero que estamos tratando entero dentro del ToS. Por ese motivo no se usa una búsqueda mediante el Tag, ya que lo que queremos es saber si existe dentro del ToS un solo nodo que complete en su totalidad el agujero. Si se usara el Tag como referencia de búsqueda podríamos encontrar un nodo perteneciente al agujero pero no habría confirmación de que esta componente encontrada formase el agujero entero, y generaría errores en la definición de las diversas componentes que conformarían dicho agujero. Si existe la coincidencia entonces se establece el parentesco entre el nodo encontrado del agujero y el nodo original que lo encapsula.

Si el agujero no está en el ToS se extiende la búsqueda usando la Lista de Adyacencia (Adj), y esta es un poco más extensa. Esta vez lo que buscamos no es el agujero, es el nodo/componente que encapsula al agujero (el nodo *father*) en la lista de adyacencia *Adj*. Una vez encontrada la componente en la lista Adj obtenemos sus vecinos. Dado que el agujero puede estar formado por varias componentes conexas distintas y todas ellas vecinas a la componente que las encapsula esta búsqueda debe ser más completa. Una vez obtenemos los vecinos de la encapsuladora se recorren los pixeles del agujero para ver si alguno de ellos coincide con algún Tag de los componentes vecinos encontrados en Adj. Si la búsqueda es certera obtendremos los nodos vecinos de la encapsuladora que pertenecen al agujero, y posteriormente su GrayLevel. Con esa información se genera un vector con todos los nodos vecinos a *father* pertenecientes al agujero y otro con el nivel de gris de dichos nodos.

A partir del hallazgo se genera una lista de las componentes encontradas pertenecientes al agujero (*component*) donde se indican sus pixeles, su valor de gris, su Tag y sus componentes vecinas del *Adj*. Todos los nodos de esta lista están ordenados de modo descendiente mediante su Valor de Gris. Cada nodo de *component* tiene su nodo original en la lista de adyacencia, el cual se marca como analizado.

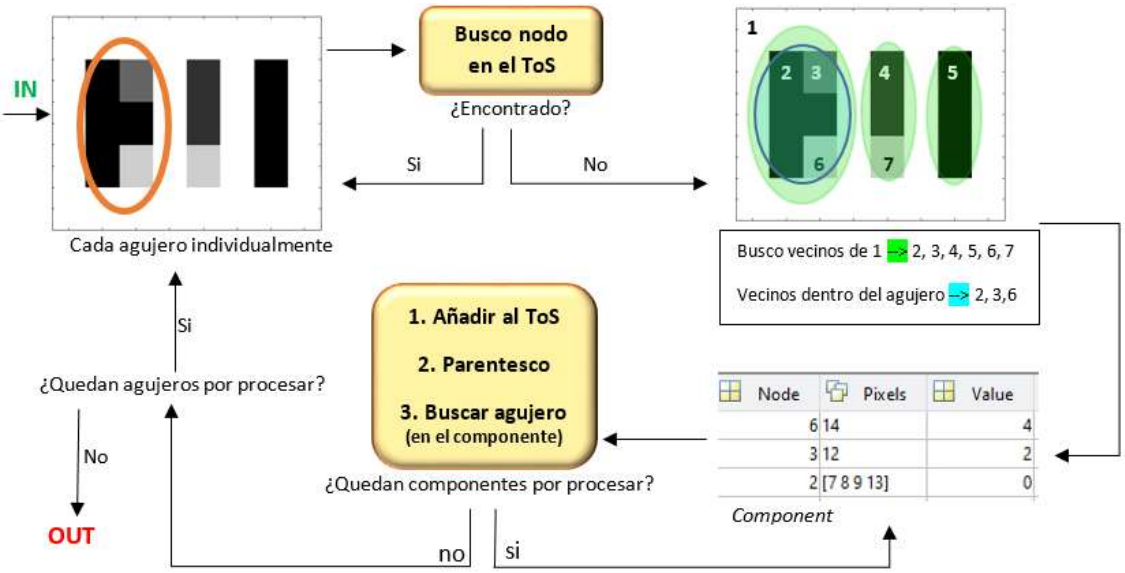

Figura 15 Algoritmo de análisis de los agujeros

Seguidamente, para cada componente del agujero encontrado se le aplica el mismo patrón (de mayor nivel de gris a menor):

- Se añade la componente estudiada a las máscaras total y de agujero como zona analizada. (*mask_total, mask_hole*).
- Se añade la componente como nuevo nodo de Tree Of Shapes.
- Se establece el parentesco entre esta componente y el nodo/componente padre (*dad*). (El nodo del agujero es hijo del nodo encapsulador solo en la primera iteración.).
- El nuevo nodo padre (*dad*) de las siguientes componentes pasa a ser la componente actual, ya que su valor de gris es superior a las componentes que le precederán.
- Finalmente, a la componente se le aplica esta misma función (*search_holes*) para buscar si esta también tiene agujeros dentro.
- Se eliminan los nodos ya analizados en *search_holes* de la lista *child* (Lista de posibles hijos de futuros nodos del ToS).
- Se procede de igual forma con la siguiente componente de la lista.

Finalmente la función retorna todas las variables de interés que han sido modificadas: *mask_total* (con los nuevos pixeles analizados introducidos), la lista *child* actualizada y el Tree of Shapes (con las nuevas componentes como nodos).

3.3.3. Algoritmo principal

Esta es la parte principal del algoritmo final, es la función que organiza todo el proceso y para ello usa las funciones anteriormente explicadas. Su función es crear en su totalidad el Tree of Shapes o Árbol de Formas.

El primer paso es generar las herramientas necesarias para generar el árbol de formas (Tree Of Shapes [ToS]) mediante la función de pre-procesado *(preprocessed*). Esta función aporta las máscaras iniciales, la Lista de Adyacencia y su orden de análisis.

El algoritmo pretende recorrer la Lista de Adyacencia para crear los nodos del Tree Of Shapes a partir de sus componentes, para eso se van a recorrer todos los nodos del Adj de mayor a menor nivel de gris (orden dictado por el vector *node_ordered*, ofrecido por *preprocessed* gracias al algoritmo Quick_Sort).

De este modo se analiza nodo a nodo la Adj siguiendo el orden establecido por *node_ordered* hasta tener todos los nodos analizados e introducidos en el ToS.

Se aplica el siguiente algoritmo a cada uno de los componentes que conforman la imagen, disgregados en la lista *Adj* y ordenados por *node_ordered*.

- Se añaden los pixeles de la componente a la máscara total (*mask_total*) y se crea una máscara de rama donde solo estará la componente y sus posibles agujeros.
- Se marca el nodo de Adj como analizado, para evitar un análisis posterior.
- Se añade la componente como un nuevo nodo del árbol de formas *Tree Of Shapes* y se registra como padre (*father*) para crear el parentesco con sus posibles agujeros internos.
- Mediante la función *search_holes* se buscan agujeros dentro de la componente. Estos agujeros serán hijos de la componente, hecho que se regula con la variable de parentesco (*father*). La función recibe la máscara de rama para hacer su cometido.
- El nodo actual del ToS se añade a la lista *child,* una lista de posibles hijos de los nodos que se analizarán en el futuro.
- Mediante la función *family_Tree* se buscan los hijos del nuevo nodo del Tree of Shapes si los tiene y estos ya han sido analizados e introducidos al ToS y por consecuencia deberán estar en la lista *child*. Como los hijos suelen ser nodos de mayor nivel de gris que el nodo analizado, y por lo tanto ya han sido analizados (orden de mayor a menor nivel de gris) es muy probables encontrarlos.
- Pasar al siguiente nodo de *Adj*.

Finalmente se aplica una nueva función para terminar de determinar algunos parentescos que no han sido detectados anteriormente, *accurate_family*.

3.3.4. Parentesco

Estas dos funciones son las encargadas de determinar la jerarquía del ToS, mediante las ramas del árbol, al decidir cuales relaciones de parentesco hay entre cada nodo. Hasta el momento tenemos nodos emparentados siempre y cuando estos sean o formen parte de un agujero, detallado en el bloque *search_holes.* Existen dos funciones con el mismo objetivo. Una de ellas es previa a la creación total del ToS y la otra es posterior, la última ejecución del algoritmo. Podría ser solo una, pero incluir la función extra de parentesco a mitad del algoritmo facilita el algoritmo y lo hace un poco más rápido porqué acota los posibles hijos.

3.3.4.1. Parentesco previo a finalización (Family_Tree)

Child-->Lista de nodos ya analizados que pueden ser hijos del nodo actual. Nodos del ToS sin padre.

Esta función se llama al final del procesamiento de cada uno de los componentes de *Adj* y por lo tanto de los nuevos nodos de Tree of Shapes. Su función es buscar hijos del nodo *father* (actual en análisis) y crear el parentesco en el árbol. Como solo buscamos en los nodos ya analizados y sin relación parental se acotan mucho las opciones para fomentar el encuentro más rápido de las relaciones de parentesco.

A partir de los posibles hijos (*child*) se genera otra lista de los Tags de esos posibles hijos (*child_tag*).

A partir del Tag del nodo *father* del Tree Of Shapes se busca su correspondencia con la Lista de Adyacencia (*Adj)* y se extraen sus vecinos. Estos vecinos pasan a formar una lista de posibles hijos del nodo father (*child_node*) y una lista de sus Tags (*child_node_tag*).

Simplemente buscando que Tags están en ambas listas sabremos el Tag de los hijos. Child_tag=child_node_tag. Una vez encontrados se establece el parentesco y los nuevos hijos de *father* se eliminan de la lista de posibles hijos (*child)* porque ya no son nodos huérfanos.

3.3.4.2. Parentesco final de algoritmo (accurate_family)

La idea es terminar de afinar el sistema de parentesco, ya que en algunos nodos no es capaz de encontrar el parentesco correspondiente ya que en el momento de usar *family_tree* no todos los nodos han sido analizados y por consecuencia no estaban en el vector de posibles hijos *child*. Eso el último paso del algoritmo, una vez terminado se considera finalizada la creación del Tree f Shapes.

Se crean dos listas, una de los nodos del Tree Of Shapes que no tienen padre y otra con sus Tags (*no_parent_node, no_parent*). Como solo puede haber un nodo sin padre (nodo referencia) y este debe tener hijos se busca el nodo de menor nivel de gris de la lista de huérfanos (*no_parent_node*) que tenga por lo menos un hijo y se elimina de la lista de huérfanos, este no debe analizarse. Esta distinción del nodo referencia como el nodo de valor de gris menor es simplemente por tener un sistema fijo de decisión, realmente no es importante el nodo raíz.

Ahora, a cada nodo huérfano se le busca en la lista *Adj* y se extraen sus vecinos (sus posibles padres, *pos_parents*) y también el nivel de gris de estos últimos. El componente dentro del vector *pos_parents* con el nivel de gris más cercano al nodo huérfano pero con un valor superior a él será su padre. Si el componente huérfano no tiene ningún vecino de nivel de gris mayor se considerará hijo del componente vecino con un valor de gris más próximo a él, aun siendo este inferior a él .Una vez encontrado el padre se busca el nodo del Tree Of Shapes que se le corresponde y se establece el parentesco.

3.3.5. Reconstrucción de la Imagen

La función de restitución de la Imagen es muy simple. La función *image_restitution_tree* recorre todo el árbol Tree of Shapes del primer nodo al último y pinta del valor de gris de cada nodo todos los píxeles asociados a dicho nodo. Una vez se haya terminado de recorrer el ToS el resultado es la imagen exactamente igual que antes de haber sido sometida al algoritmo. La función necesita saber el tamaño de la imagen para poder generarla ya que es una información omitida en el árbol.

4. Resultados

En este apartado del proyecto se van a analizar los resultados del algoritmo de creación del Tree of Shapes diseñado a través de varias imágenes con diversidad de complejidades. Se analizaran varias imágenes a partir de sus resultados y su tiempo de cálculo.

Con la función *maxtree_to_gv*, entregada por la UPC y el profesor Philippe Salembier, podemos generar un archivo (en lenguaje .dot) que el programa gráfico de software libre Graphviz (http://www.graphviz.org/) será capaz de leer y posteriormente, a partir de él, generar el esquema gráfico del Tree of Shapes visualmente. Esta función fue diseñada para visualizar los árboles Maxtree y Mintree, pero como la variable Tree of Shapes tiene una estructura muy similar a la de los árboles de extremos también puede ser visualizada mediante la misma función. La visualización es muy intuitiva, cada nodo se corresponde por una bola de un color igual al valor de gris de ese nodo y estos están relacionados mediante flechas según su jerarquía y relación de parentesco (los padres apuntan a sus hijos con la flecha).

4.1. Creación del Tree of Shapes

En estos primeros ejemplos queremos ver si realmente el algoritmo genera el árbol correctamente así que se muestran las imágenes analizadas y su árbol asociado (Visualizado mediante Graphviz). Como información adicional se añade para cada ejemplo una tabla con parámetros de complejidad de cada imagen y el tiempo de cálculo del algoritmo.

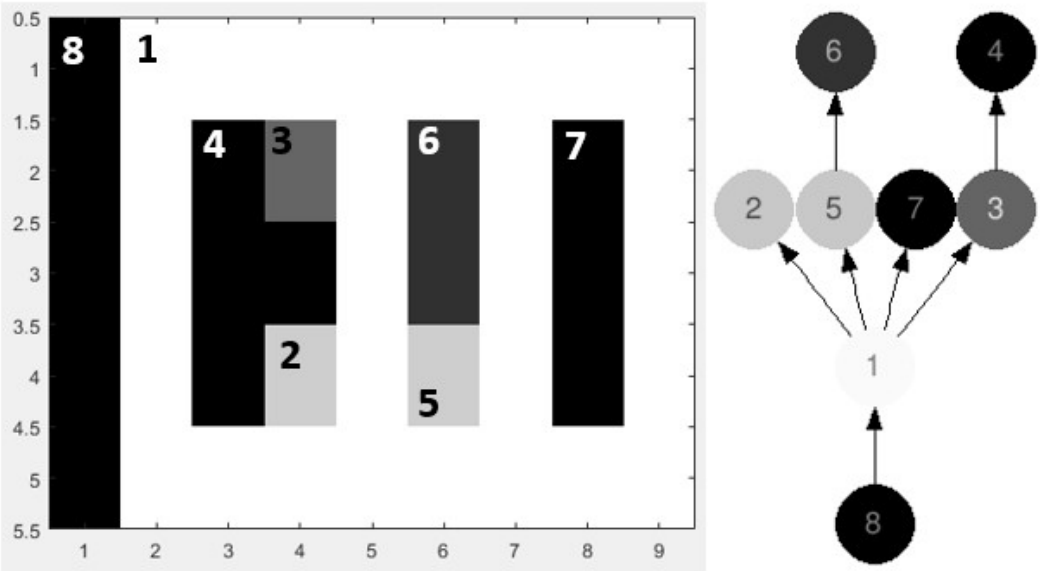

Figura 16 Imagen ejemplo 1 con su Tree of Shapes resultante

Tamaño de Imagen	Nº Gray Level	Nº Componentes	Tiempo cálculo
5x9	5	8	0.417 seg

Tabla 2 Parámetros del algoritmo en la imagen ejemplo 1

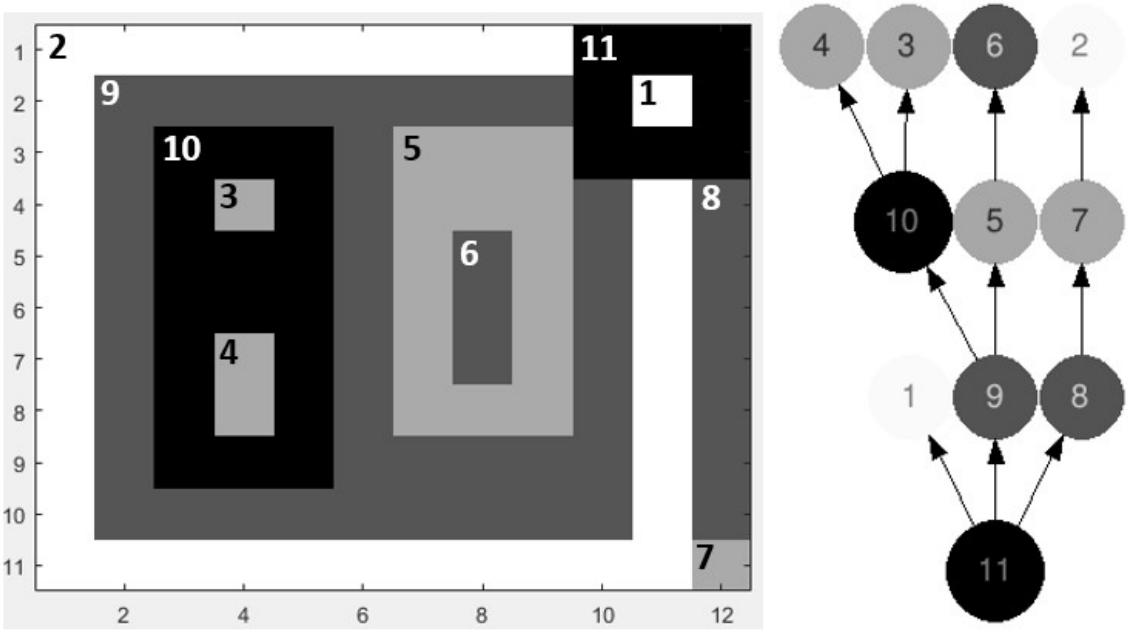

Figura 17 Imagen ejemplo 2 con su Tree of Shapes resultante

Tamaño de Imagen	Nº Gray Level	Nº Componentes	Tiempo cálculo
11x12	4	11	0.759 seg

Tabla 3 Parámetros del algoritmo en la Imagen ejemplo 2

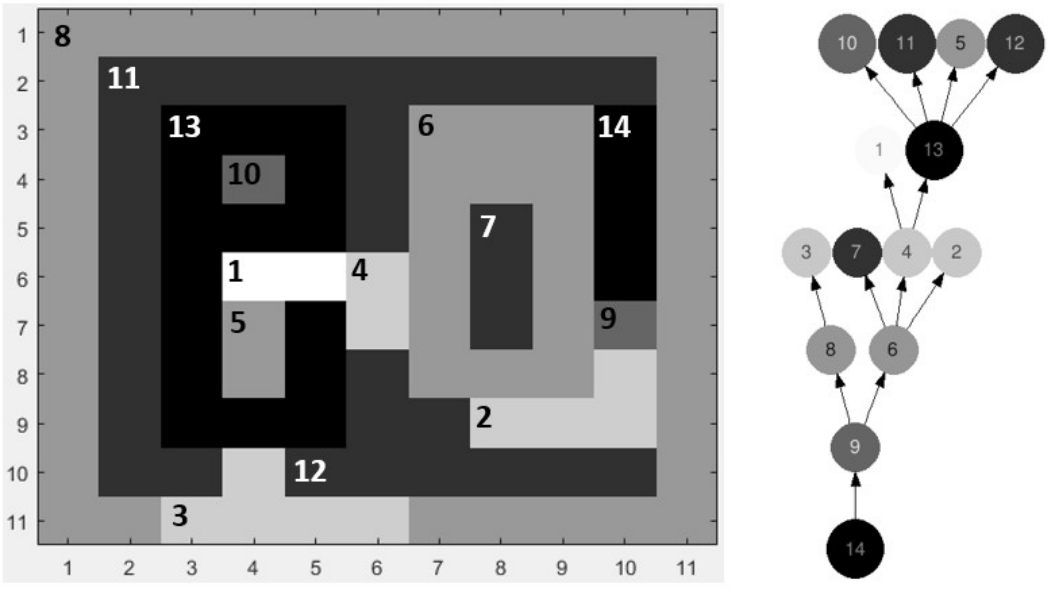

Figura 18 Imagen ejemplo 3 con su Tree of Shapes resultante

Tamaño de Imagen	Nº Gray Level	Nº Componentes	Tiempo cálculo
11x11	6	14	1.072 seg

Tabla 4 Parámetros del algoritmo en la Imagen ejemplo 3

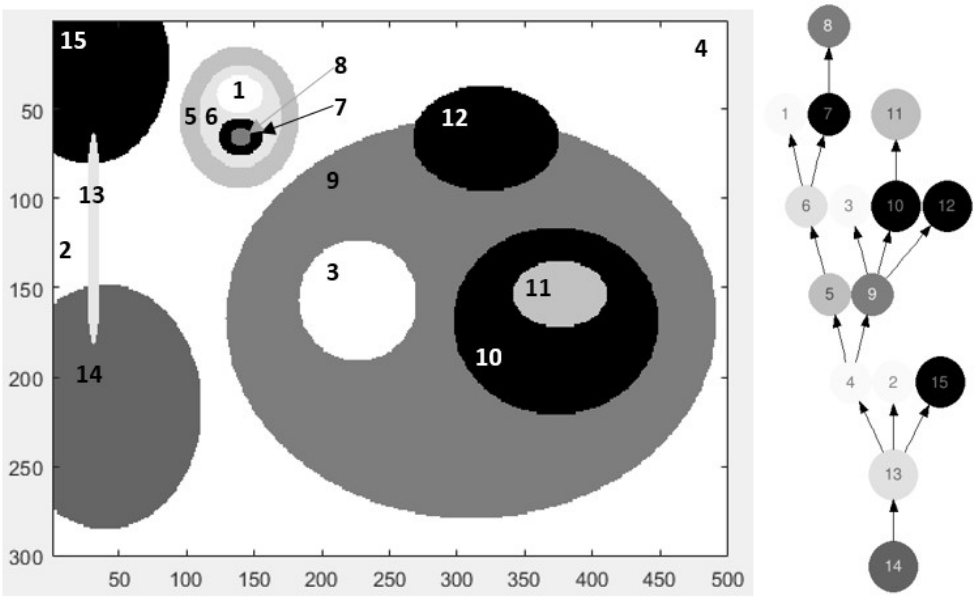

Figura 19 Imagen ejemplo 4 con su Tree of Shapes resultante

Tamaño de Imagen	Nº Gray Level	Nº Componentes	Tiempo cálculo
300x500	6	15	28.0183 seg

Tabla 5 Parámetros del algoritmo en la Imagen ejemplo 4

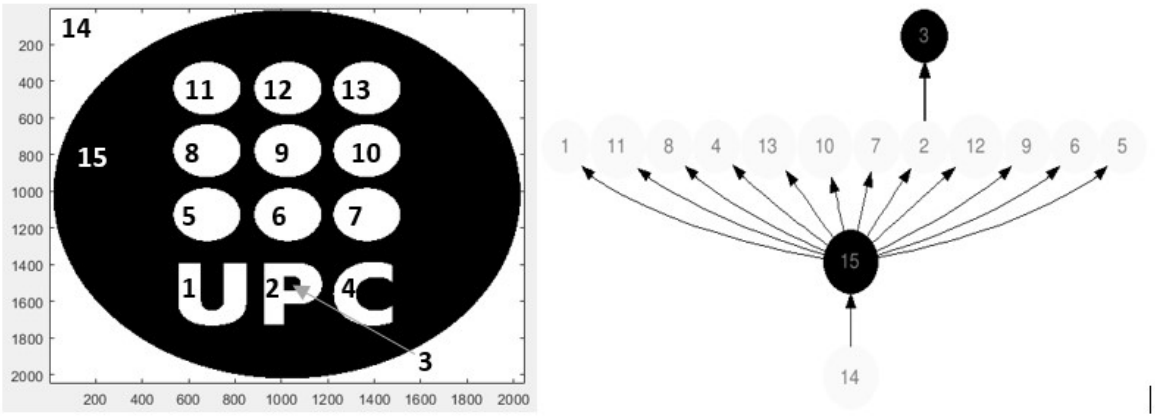

Figura 20 Imagen ejemplo 5 (Binaria) con su Tree of Shapes resultante

Tamaño de Imagen	Nº Gray Level	Nº Componentes	Tiempo cálculo
2048x2048	2	15	215,75 seg

Tabla 6 Parámetros del algoritmo en la Imagen ejemplo 5

Los ejemplos visualizados tienen la peculiaridad de que cada ejemplo es más complejo (en costes de cálculo) que el anterior. La complejidad en estos ejemplos viene dada principalmente por el tamaño de imagen, ya que el número de componentes que forman las imágenes es bajo (Todas entre los 8-15 componentes conexos). En este caso todos los ejemplos tienen un árbol de formas simple y con pocos nodos. Aun así se puede apreciar el correcto funcionamiento de esta creación. Cada árbol está construido correctamente, no contiene información duplicada y tiene un nodo para cada componente conexa de la imagen.

4.2. Diferencias del Tree of Shapes según la conectividad

Para valorar el efecto de los dos tipos de conectividad en la imagen, se añaden un par de ejemplos sencillos donde se pueden apreciar las notables diferencias en la forma del árbol en conectividad de valor 4 y de valor 8.

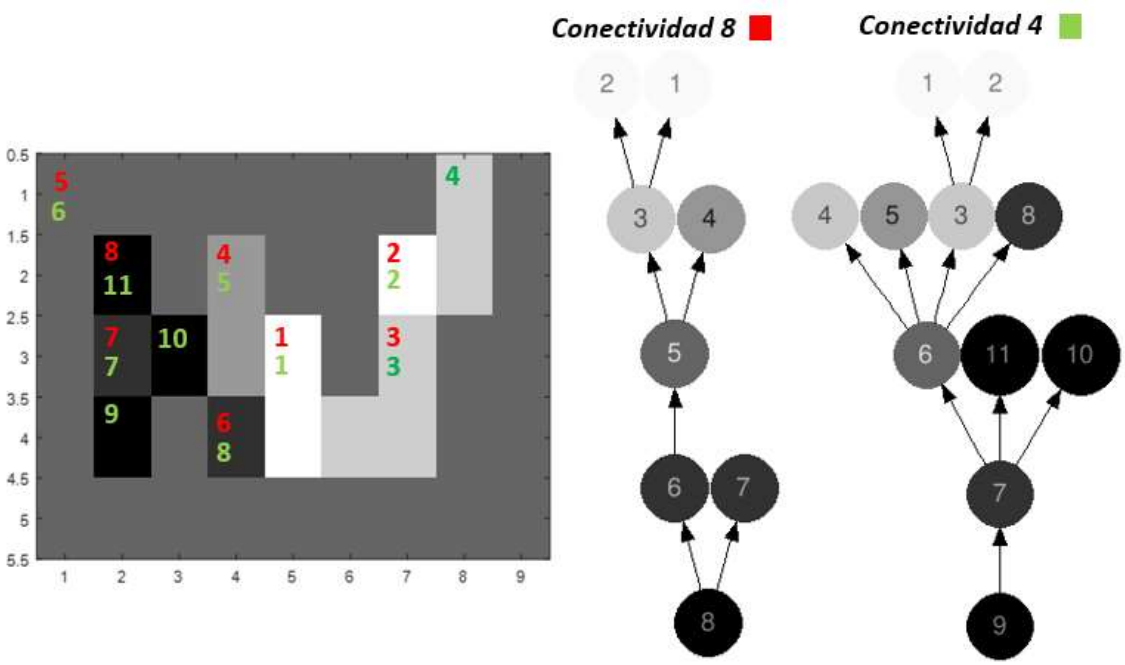

Figura 21 Imagen ejemplo Con4 con sus árboles asociados a los dos valores distintos de conectividad

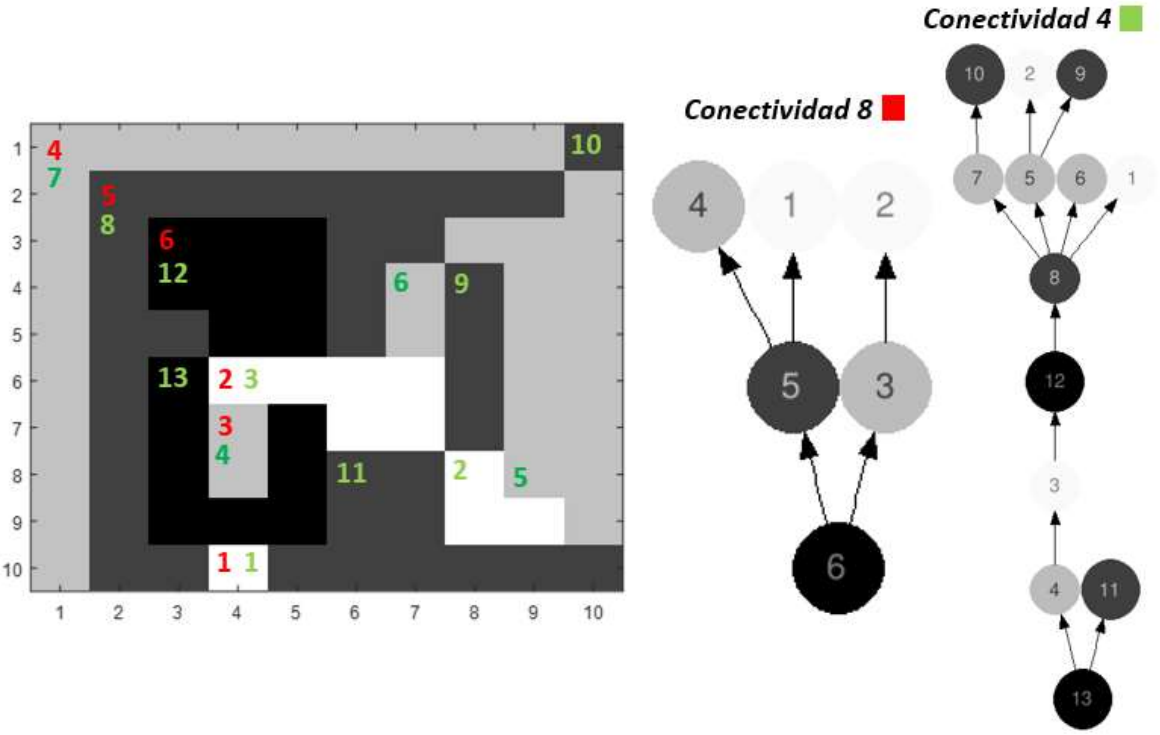

Figura 22 Imagen ejemplo Con8 con sus árboles asociados a los dos valores distintos de conectividad

En los ejemplos de la Figura 21 y la Figura 22 podemos apreciar las notables diferencias de confección del Tree of Shapes dependiendo de si el valor de la conectividad es 8 o 4. Recordemos que en las imágenes analizadas en con8 se interpretan los píxeles del mismo valor de gris que se tocan diagonalmente $[I(i,j) \leftrightarrow I(i+1,j+1)]$ como una misma componente conexa. En con4, por lo contrario, este mismo escenario se interpreta como distintas componentes y por consecuencia el número de componentes y de nodos va a ser siempre igual o mayor que en las imágenes en con8. En el Tree of Shapes de la Figura 21 en conectividad 8 solo aparece un nodo perteneciente al valor de gris mínimo en la imagen. Si miramos el árbol generado con conectividad 4 observamos que ahora hay 3 nodos pertenecientes al mismo valor de gris anterior. Lo que anteriormente en con8 se interpretaba como una solo componente, en con4 se interpreta como 3 componentes distintas, ya que entre ellas solo existe el contacto diagonalmente. En el ejemplo de la Figura 22 ocurre exactamente lo mismo, el nodo 6 del ToS corresponde a la zona de valor de gris mínimo en conectividad 8. Si miramos en el árbol con conectividad 4 esta zona está dividida en las componentes 12 y 13. Esta distinción en diferentes componentes que ocurre en los análisis con conectividad 4 es la que provoca que para las imágenes con4 siempre haya un número de componentes superior (en el mejor de los casos será igual) a los análisis en con8. Este hecho provoca que el tiempo de cálculo en conectividad 4 sea también mayor que en con8 porque debe analizar más nodos. A continuación se realizara un estudio más afondo de estas diferencias de costes computacionales.

4.3. Análisis de los costes computacionales

Ahora se debe poner énfasis en el tiempo de cálculo. Como vemos en los ejemplos anteriores existen diferencias muy notables entre el tiempo de cálculo de los últimos ejemplos (Figura 20 y Figura 19) y todos los tiempos relacionados con los primeros ejemplos (Figura 16, Figura 17 y Figura 18). Esta diferencia es notable teniendo en cuenta que no son imágenes con un número de componente elevado, por lo tanto gran parte del retraso en el tiempo de cálculo se debe a sus diferencias de tamaños. Un ejemplo clarísimo es el de la Figura 20 con un tamaño muy superior al resto (2048x2048) y con un coste computacional de más de 3 minutos (215 seg). Si la comparamos con la segunda imagen más grande (Figura 19 con tamaño 300x500) se ve evidente que esta necesita mucho menos tiempo para ejecutar el algoritmo (28 seg) aun teniendo el mismo número de componentes (15). Por consecuente lo normal será esperar un gran aumento del tiempo de cálculo para imágenes más complicadas con más componentes y un rango de valores de gris superior.

A partir de aquí el algoritmo es considerado correcto en referencia a su funcionamiento de creación del Tree of Shapes, y se estudiará los costes temporales que requiere el algoritmo para ejecutarse en diversos escenarios. Se emplearan diversas imágenes ejemplos con varios grados de dificultad.

Usando la imagen 'Cameraman.png' se analizan los distintos tiempos de cálculo que precisa el algoritmo para distintas calidades y complejidades de la imagen. La complejidad de la imagen se valora mediante el rango de valores de gris que contiene y especialmente por la cantidad de componentes conexas a analizar. Para tener distintas calidades de imagen y consecuentemente distintas complejidades se aplicará a la imagen una cuantificación uniforme en función del valor N. Cuanto mayor sea el valor N menos complejidad y de menos componentes estará integrada la imagen. $I_{Norm} = N * round\left(\frac{I(i,j)}{N}\right) \forall\, i,j \in I$. El tamaño de la imagen es fijo en todos estos casos, de tamaño cuadrado 229x229.

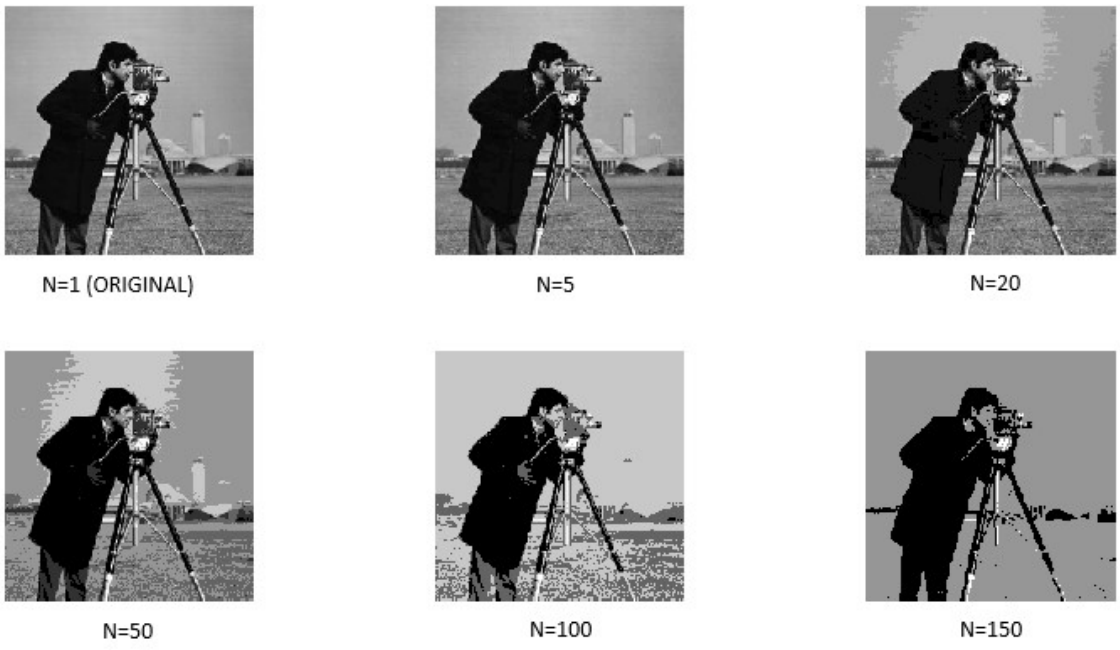

Figura 23 Cameraman con distintas calidades según el valor de N

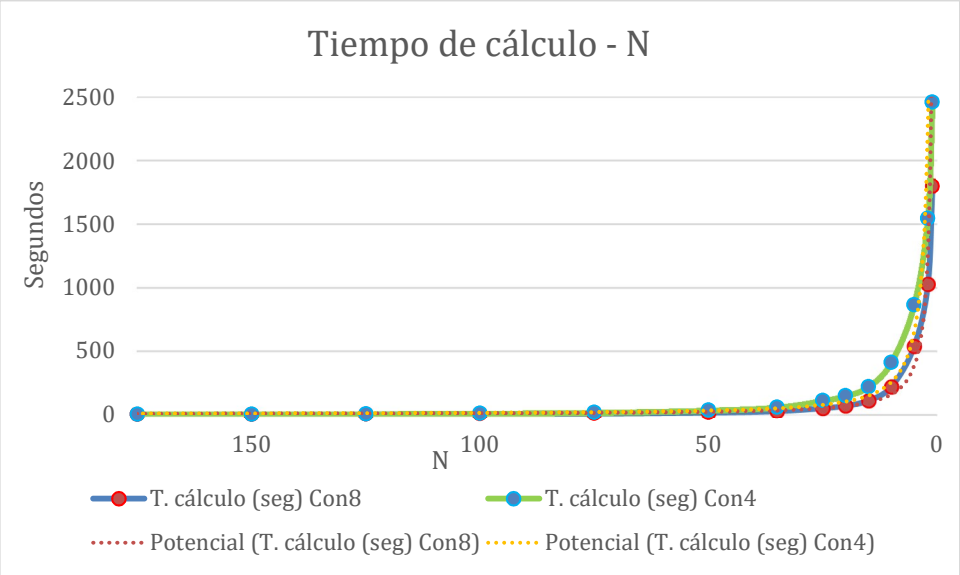

Figura 24 Coste computacional en tiempo (seg) de la Imagen Cameraman a distintos valor de N para conectividad 4 y conectividad 8

Podemos observar en la Figura 24 cómo se cumplen las predicciones. Cuanto mayor es el valor N menos complejidad en la imagen y por lo tanto su tiempo de cálculo es sensiblemente menor. Para valores bajos de compresión (más próxima a la imagen original, N menor) el coste computacional aumenta enormemente. También se aprecia que para los mismos valores de compresión de la imagen en los casos de conectividad 4 y conectividad 8, los tiempos de cálculo pertenecientes al algoritmo en con4 son mayores que en con8 debido a su número superior de componentes conexas. Este hecho se hace más evidente cuanta mayor complejidad tiene la imagen (en valores inferiores de compresión según N).

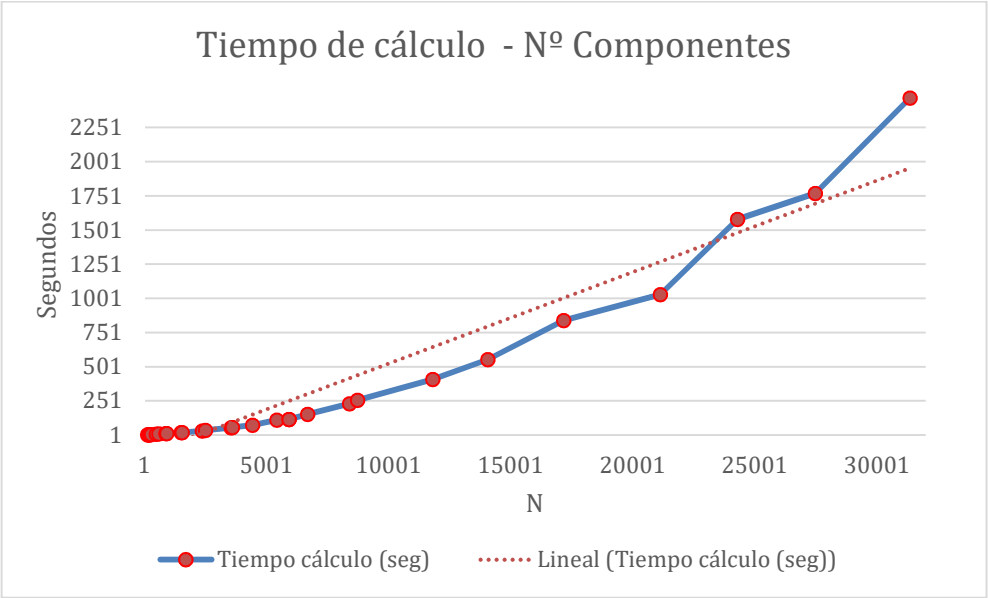

Figura 25 Coste computacional en Tiempo (seg) en relación a las componentes estudiadas para la Imagen Cameraman

La Figura 25 analiza el coste computacional que precisa el algoritmo dependiendo de la cantidad de componentes conexas que deba procesar. Aquí no se distingue entre distintas conectividades, realmente lo que provoca una conectividad u otra es que existan más o menos componentes. Se pueden observar en la Figura 25 resultados obvios que se pueden extraer viendo la Figura 24, cuanta más complejidad y cuanto mayor sea el número de componentes que conforman la imagen más tiempo precisará el algoritmo para generar el Tree of Shapes.

El mismo estudio se procede a partir de la imagen 'Lena.bmp' aplicando la misma función cuantificadora (en función de N). Dicha imagen tiene un tamaño superior a la anterior y previsiblemente se esperan tiempos de cálculo superiores a los del Cameraman. Lena es una imagen cuadrada de tamaño 512x512.

Figura 26 Lena con distintas calidades según el valor de N

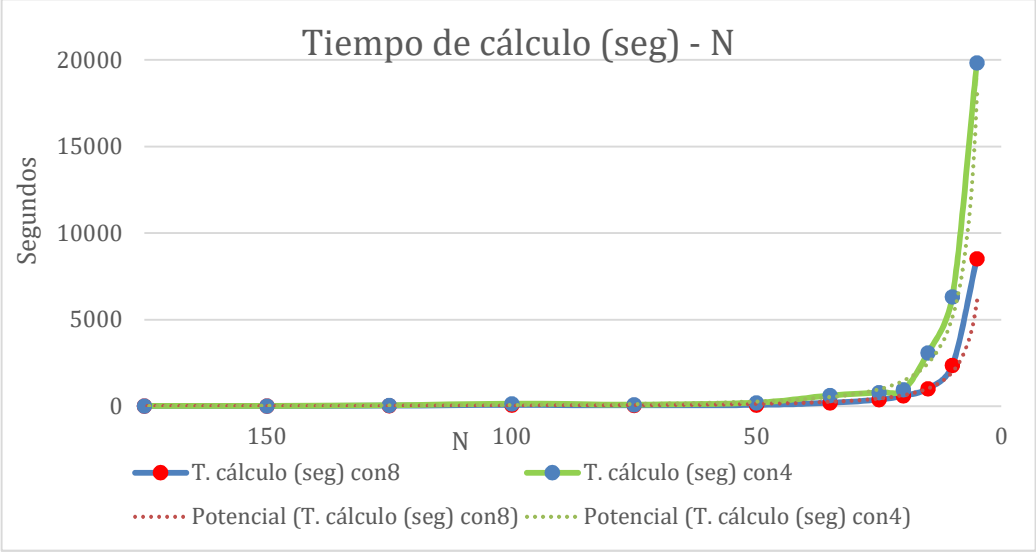

Figura 27 Coste computacional en tiempo (seg) de la Imagen Lena a distintos valor de N para conectividad 4 y conectividad 8

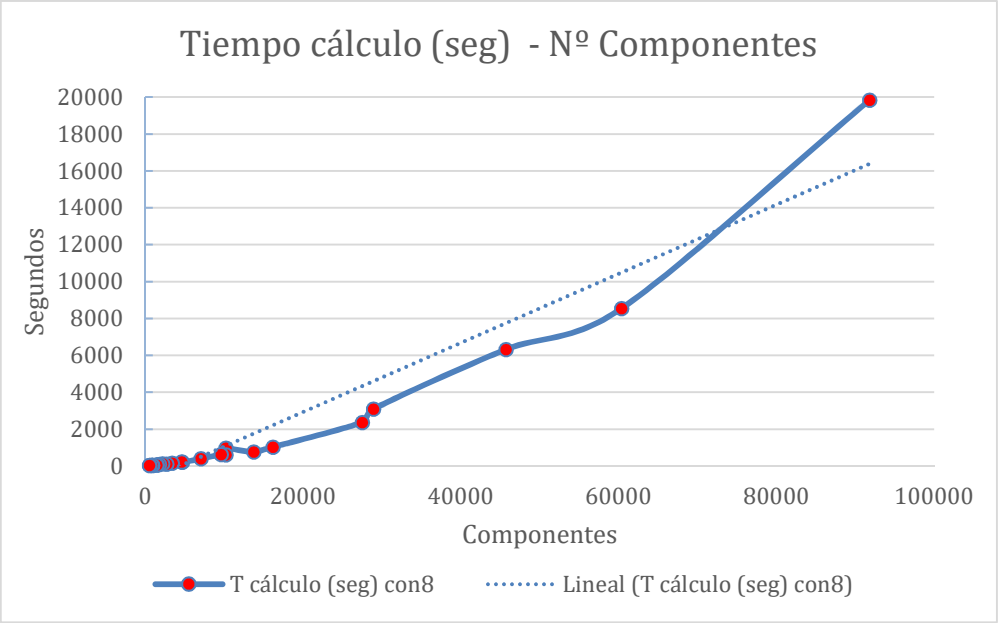

Figura 28 Coste computacional en Tiempo (seg) en relación a las componentes estudiadas para la Imagen Lena

Con el ejemplo Lena observamos la misma tendencia anterior, el tiempo invertido por el algoritmo es mayor cuanta más complejidad y más componentes conexas tenga la imagen. De nuevo se aprecia que los análisis en conectividad 4 precisan de un coste computacional superior. En el caso de la Figura 27 se observa un incremento del tiempo de cálculo en contraposición con el anterior ejemplo Cameraman [Figura 24Figura 24] para los mismos valores de N. Pero si comparamos las componentes de la imagen Lena [Figura 28] con la Figura 25 de Cameraman es fácil percatarse de que Lena tiene muchas más componentes conexas. Este aumento de componentes implica que la imagen Lena es más compleja que Cameraman y su coste computacional en tiempo es superior. También es importante fijarse en el tamaño de Lena, 512x512; que al ser mayor que la imagen Cameraman, 229x229; también hace aumentar el tiempo de cálculo susceptiblemente. A demás cuanto mayor es la imagen más posibilidad existe que contenga más componentes conexas que una imagen de tamaño menor.

Si interpretamos los gráficos de coste temporal relacionados con el número de componentes conexas [Figura 25 y Figura 28] podemos diferenciar cierto comportamiento lineal de los datos (podemos comparar con su línea de tendencia lineal). El tiempo total de cálculo desempeñado por el algoritmo está estrechamente relacionado con el número de componentes que se deben estudiar, y siguen una relación que se puede aproximar como lineal. Este comportamiento aproximadamente lineal descarta que el retraso temporal del programa se deba a que el algoritmo le cuesta tratar con cantidades altas de componentes ya que emplea aproximadamente el mismo tiempo por componente (tiempo/componente). Si el motivo no es que se acelera el retraso con el número de componentes es porque dicho motivo está relacionado con la calidad de la imagen. Aunque el coste temporal sea proporcional a los componentes, la cantidad de componentes no es lineal a su factor de reducción de calidad. Cuanta más compresión aplicamos a la imagen más se acelera el descenso del número de componentes. Si lo tomamos desde otra perspectiva, cuanta más calidad tenga una imagen, (menos compresión) más rápido aumentaran su cantidad de componentes conexas. En los ejemplos anteriores [Figura 24 y Figura 27] podemos examinar este hecho. Viendo que tanto en conectividad 4 como 8, a medida que la calidad de imagen aumenta, aumenta también el factor de crecimiento del coste computacional en

tiempo. Este crecimiento sigue un patrón de crecimiento potencial y cuanta mayor calidad estemos analizando mayor crecimiento sufrirá el tiempo de cálculo. Si contemplamos la relación del coste temporal y el número de componentes como lineal [Figura 25 y Figura 28], aparte de afirmar que el aumento de la calidad de la imagen hace agrandar aceleradamente el tiempo de cálculo también podemos afirmar que aumentará el número de distintas componentes conexas con el mismo factor de crecimiento, y este aumento de las componentes es realmente el que provoca que el tiempo de cálculo aumente. Que la compresión sea menor implica que la imagen tendrá más componentes conexas distintas, y ellas provocan su vez que los costes computacionales aumenten. Estos costes aumentan al orden de una función potencial como podemos ver en la línea de tendencia de la Figura 29.

A continuación se puede observar como la cantidad de componentes conexas aumenta a medida que la compresión es menor (N próxima a 1). El crecimiento del número de componentes sigue una tendencia muy similar al crecimiento del tiempo de cálculo, hecho que reafirma la relación lineal entre ambas [Figura 29].

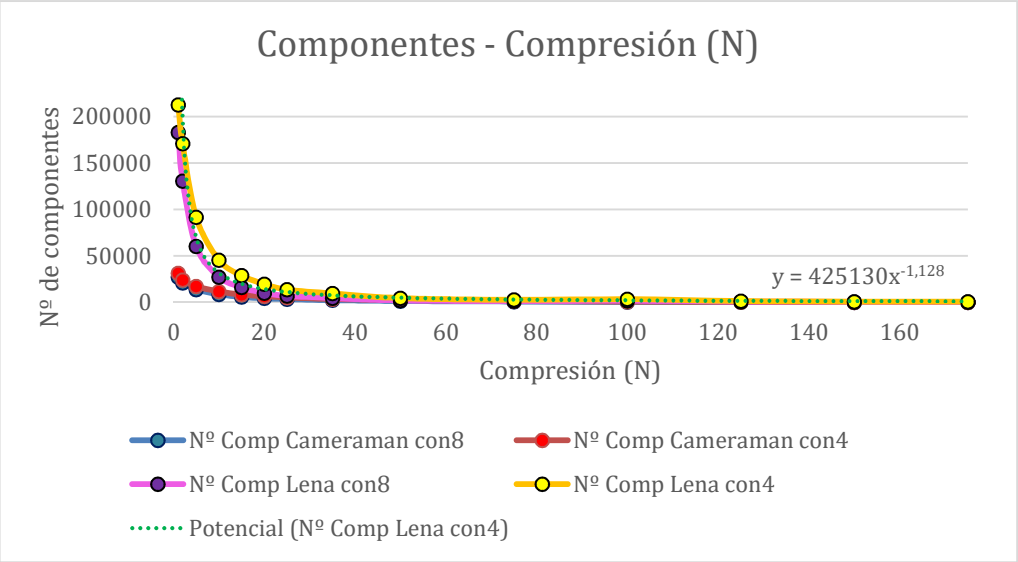

Figura 29 Relación entre el valor de compresión N y el número de componentes de Lena y Cameraman para las distintas conectividades

Vemos como a medida que la calidad de la imagen aumenta por el descenso del valor de la variable N, [Figura 29] la cantidad de componentes crece con un factor de crecimiento acelerado. Este comportamiento del aumento de las componentes es el que provoca que haya los costes temporales se disparen para imágenes más complejas.

4.3.1. Análisis de los costes computacionales por función

Finalmente quiero hacer un inciso en cuáles son las funciones del algoritmo que más tempo consumen.

Las funciones que consumen mucho coste de computación son las funciones más esenciales del algoritmo, como las encargadas de crear las relaciones entre nodos (*family_tree [3.3.4.1]* y *accurate_family [3.3.4.2]). Otra función que consume un tiempo de cálculo muy considerable es search_holes [3.3.2],* que busca agujeros en los nodos del Tree of Shapes. La otra función importante es la primera del algoritmo (*pre_processed [3.3.1]*). Esta última es especialmente pesada a mayor tamaño de la imagen y no es tan dependiente de la complejidad como las otras. Las funciones mencionadas son totalmente imprescindibles, y en una posible mejora del algoritmo habría de trabajar en optimizarlas para que tengan un desarrollo mucho más veloz y menos costoso computacionalmente. El cuanto al conjunto final del tiempo de cálculo parece un hecho difícil de obviar que consume demasiado tiempo para convertirse en un modelo realmente útil. Para imágenes grandes y con una complejidad alta con un gran número de componentes conexas parece insalvable intentar mejorar el algoritmo para reducir sensiblemente el algoritmo.

Para conocer el peso que tiene cada una de estas funciones en el tiempo total de cálculo podemos observarlo en las gráficas Figura 30 y Figura 31. Vemos que peso temporal en referencia al tiempo total de cálculo está poco relacionado con la conectividad, ya que ambas medidas son notablemente parecidas. Las funciones más pesadas globalmente son las encargadas de crear las relaciones entre nodos (*family_tree* y *accurate_family*), que especialmente cuando la complejidad de la imagen aumenta (más componentes a estudiar) también aumentan su importancia temporal en el algoritmo. Especialmente *family_tree*, que es la función que genera un coste computacional superior. El caso contrario serían las funciones *pre_processed* y *search_holes*, que a medida que la imagen se vuelve más compleja pierden peso en su coste computacional. Estas últimas funciones son especialmente influyentes en las imágenes más simples del espectro de complejidad. Estas funciones clave del algoritmo ocupan prácticamente la totalidad del tiempo de cálculo. Si vemos la línea Tiempo Total en observamos en ambos casos como la suma de los porcentajes de las 4 funciones anteriores pertenecen a más de 90% del tiempo total de cálculo para todas las complejidades.

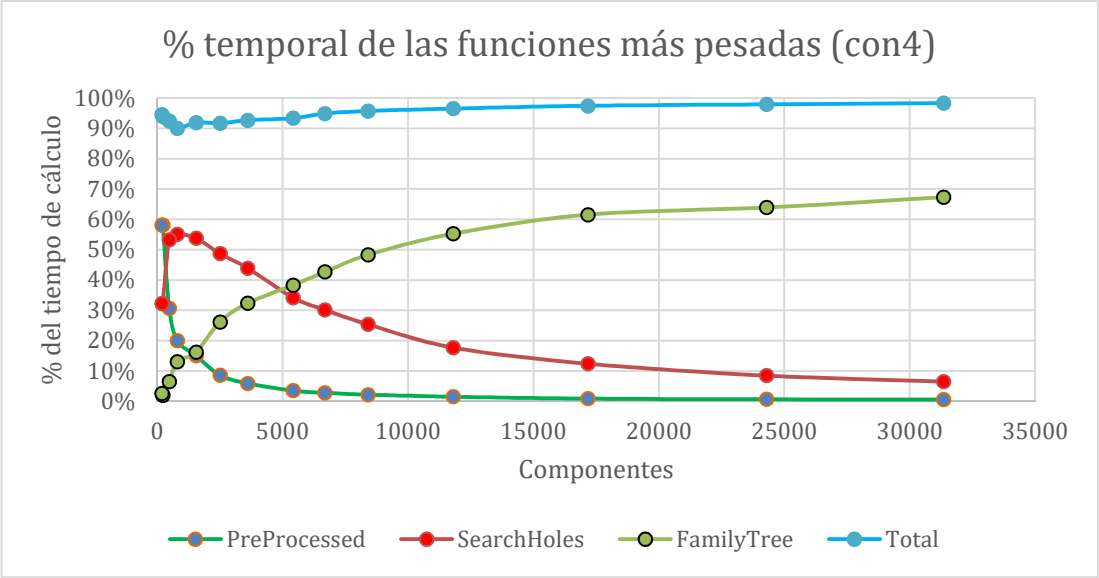

Figura 30 Peso de las funciones principales en el tiempo de cálculo total del algoritmo en conectividad 4

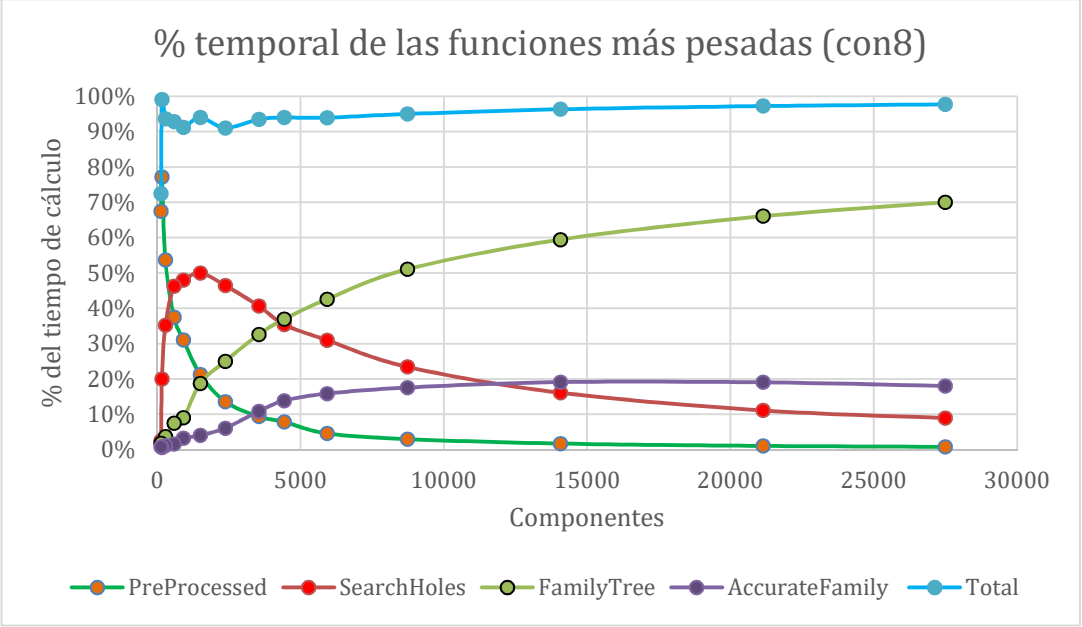

Figura 31 Peso de las funciones principales en el tiempo de cálculo total del algoritmo a conectividad 8

5. Presupuesto

En este capítulo se realiza una estimación del coste total del proyecto realizado. Dichos costes pueden ser divididos en tres partes (Tabla 7)

- **Salario del personal**
  El ingeniero encargado del proyecto es solo una persona y con el estatus de ingeniero júnior por terminar recientemente su licenciatura. Por ese motivo se le retribuye con un salario de 12€/h. La duración de proyecto ha sido de 32 semanas con 5 días laborables cada una aproximadamente (ha habido 6 días festivos durante el desarrollo del proyecto) y 4 horas de trabajo diarias.
- **Software**
  Simplemente conseguir la licencia Matlab. Como estudiante la Licencia es gratuita en la universidad UPC. Como empresa hay que pagar la licencia, y en este supuesto se ha decidido una licencia anual.
- **Hardware**
  Material de trabajo básico para la realización del trabajo es un ordenador capaz de sobrellevar una gran cantidad de cálculo. Se ha seleccionado un ordenador portátil por comodidad, y es la versión HP 15-BS090NS. Es un ordenador de precio moderado y de buenas prestaciones.

Elemento	Unidades	Precio
Salario		
Semanas trabajadas	32	
Días laborables/Semana	5	
Horas diarias	4	
Salario júnior por hora	1 hora	12 €
Días de Trabajo	154	
Horas de Trabajo	616	
Total Salario		7.392 €
Software		
Matlab Standard License	1 año	800 €
Hardware		
Portátil HP 15-BS090NS	1	699 €
Gasto Total		**8.891 €**

Tabla 7 Coste económico del proyecto

6. Conclusiones y líneas futuras

6.1. Conclusiones

El objetivo principal de este proyecto era el diseño, construcción e implementación de un sistema de representación de una imagen basado en un árbol jerárquico, el árbol de formas Tree of Shapes.

La idea inicial es crear el ToS usando los arboles de máximos y mínimos, una idea básica parecida a la del profesor Monasse. Con el estudio de los árboles Maxtree, Mintree y Tree of Shapes de Monasse se avanza en la confección del algoritmo.

En pleno proceso de creación se descubre una problemática que se soluciona mediante una variación del algoritmo, que termina generando un algoritmo más complejo y costoso computacionalmente. La idea de generar un nodo para cada componente conexa de la imagen empieza a apartarse ligeramente de la idea de Tree of Shapes del profesor Monasse. Una vez diseñado todo el algoritmo se percibe una idea de que no es óptimo debido al uso del Maxtree y Mintree, ya que en la última versión del algoritmo generaban más problemáticas que beneficios.

Con este hecho se decide generar otro algoritmo distinto, siguiendo una estructura parecida y aprovechando funciones anteriores, pero distinto y sin el uso de los árboles de extremos Maxtree y Mintree. Este finalmente si es el algoritmo definitivo del proyecto. Genera un árbol de formas distinto al de Monasse pero que expresa la misma idea de relaciones y jerarquías entre zonas planas de la imagen. Al igual que en algoritmo de Monasse, los agujeros juegan una parte importantísima del ToS, pero como novedad, en vez de relacionar los nodos mediante las relaciones establecidas por Maxtree y Mintree se establecen mediante el vecindario del nodo correspondiente en la imagen original.

A la hora de visualizar los resultados parece indicar que el algoritmo funciona correctamente. Todas las componentes y zonas de la imagen tienen su nodo correspondiente y están emparentadas con sus nodos vecinos o sus nodos agujeros, dependiendo del caso. No hay información redundante en el árbol que era una especificación fundamental. Y la reconstrucción de la imagen original a partir del árbol es perfecta sin ningún tipo de pérdida o modificado de información. La imagen resultante es idéntica a la imagen original.

Aun así, hay que mencionar el elevado coste temporal del algoritmo. Para imágenes de tamaño grande el tiempo de cálculo supera con creces el tiempo razonable para ser un algoritmo útil. Y parece poco servible usar el algoritmo solo en imágenes pequeñas o con baja complejidad.

6.2. Líneas futuras

El paso más importante que se debe dar es rebajar el tiempo que el algoritmo necesita para ejecutarse. Como se muestra anteriormente (Figura 30 y Figura 31) son 4 las funciones principales que generan el retraso. Aun así la función de pre-procesado no tiene mucha importancia para grandes tamaños y complejidades y poco margen de mejora puede tener. Por lo tanto, las futuribles mejoras se deberían centrar en 3 funciones.

- Crear una función de búsqueda de agujeros más veloz sería una ayuda importante para rebajar los costes computacionales. No es la función más pesada del programa, pero si la que tiene margen de mejora considerable. Una de las posibles soluciones sería la mejora de las máscaras. Si se usaran máscaras adaptadas al tamaño de los componentes estudiados y no máscaras del tamaño total de la imagen se podría reducir el tiempo de cálculo para cada iteración de *search_holes*, y como es una función muy concurrida haría descender el tiempo total de cálculo del sistema.
- Las dos funciones de relación (*family_tree* y *accurate_tree*) son las principales pérdidas de tiempo de cómputo. Aun así, la creación de family_tree reduce el tiempo total al existir, porqué si solo se trabajase con *accurate_tree* sería aún más lento el algoritmo. Estas dos funciones deberían ser estudiadas más a fondo y buscar alternativas más veloces de funcionamiento.

Otras mejoras que podría tener el algoritmo es la inclusión de la posibilidad de generar los nodos con una cierta tolerancia de umbral. Crear un umbral variable ($h = x \pm \Delta$). Pero hay que tener en cuenta que si se produce esta tolerancia de umbral, la imagen resultante de la reconstrucción del árbol no sería la misma que la imagen original, tendría un aumento en las pérdidas cuanto mayor sea la tolerancia Δ. Sería interesante tener la opción de simplificar el árbol a partir de la imagen original. Hasta el momento tenemos la opción de simplificar la imagen mediante la cuantificación uniforme de valor N. Si se añadiese este factor de tolerancia en el umbral, se podría simplificar la imagen y el árbol resultante a la vez teniendo más opciones en sus futuras funcionalidades.

Bibliography

[1] Philippe Salembier, Albert Oliveras, Luis Garrido. "Antiextensive Connected Operators for Image and Sequence Processing". *IEEE Transactions on Image Processing*, vol. 7, no. 4, pp. 555-570, April 1998.

[2] Pascal Monasse, Frédéric Guichard. "Fast Computation of a Contrast-Invariant Image Representation". *IEEE Transactions on Image Processing*, vol. 9, no. 5, pp. 860-872, May 2000.

[3] Pascal Monasse. “Morphological representation of digital images and application to registration”. Doctoral thesis, Université Paris IX-Dauphine, 30 June 2000.

[4] Edwin Carlinet. “Un Arbre de Formes Pour des Images Multi-Variées a Tree of Shapes for Multivariate Images”. Doctoral thesis, Université Paris-Est, 27 November 2015.

[5] “QuickSort Debugging in Matlab”, https://stackoverflow.com/questions/15047606/quicksort-debugging-in-matlab, 2013.

Glosario

ToS	Tree of Shapes
FLST	Fast Level Set Transform
FLLT	Fast Level Lines Transform